Airblue: A Highly-Configurable FPGA-Based Platform for Wireless Network Research

by

Man Cheuk Ng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Arvind
Johnson Professor
Thesis Supervisor

Accepted by .....
Leslie Kolodziejski
Chairman, Department Committee on Graduate Students

Airblue: A Highly-Configurable FPGA-Based Platform for Wireless Network Research

by

Man Cheuk Ng

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Over the past few years, researchers have developed many *cross-layer* wireless protocols to improve the performance of wireless networks. Experimental evaluations of these protocols require both high-speed simulations and real-time on-air experimentations. Unfortunately, radios implemented in pure software are usually inadequate for either because they are typically two to three orders of magnitude slower than commodity hardware. FPGA-based platforms provide much better speeds but are quite difficult to modify because of the way high-speed designs are typically implemented by trading modularity for performance. Experimenting with cross-layer protocols requires a flexible way to convey information beyond the data itself from lower to higher layers, and a way for higher layers to configure lower layers dynamically and within some latency bounds. One also needs to be able to modify a layer's processing pipeline without triggering a cascade of changes. In this thesis, we discuss an alternative approach to implement a high-performance yet configurable radio design on an FPGA platform that satisfies these requirements. We propose that all modules in the design must possess two important design properties, namely *latency-insensitivity* and *data-driven control*, which facilitate modular refinements. We have developed *Airblue*, an FPGA-based radio, that has all these properties and runs at speeds comparable to commodity hardware. Our baseline design is 802.11g compliant and is able to achieve reliable communication for bit rates up to 24 Mbps. We show in the thesis that we can implement SoftRate, a cross-layer rate adaptation protocol, by modifying only 5.6% of the source code (967 lines). We also show that our modular design approach allows us to abstract the details of the FPGA platform from the main design, thus making the design portable across multiple FPGA platforms. By taking advantage of this virtualization capability, we were able to turn Airblue into a high-speed hardware software co-simulator with simulation speed beyond 20 Mbps.

Thesis Supervisor: Arvind
Title: Johnson Professor

Acknowledgments

After nearly a decade, my journey to the Ph.D is close to the end. I owe a debt of gratitude to many people who helped me to complete this lengthy yet smooth ride.

First and foremost, I thank Prof. Arvind, my research supervisor, for giving me the freedom and the financial support to work on any project. Prof. Arvind is very unique at supervising his students. He does not tell us what to do. Instead, he encourages us to explore different topics ourselves. As a result, students in his group are usually involved in multiple projects throughout their studies. I myself have worked on multiple projects including the designs of cache coherence engines, network-on-chips and wireless basebands. I appreciate his approach because it has helped me to broaden my horizon.

I thank Prof. Joel Emer and Prof. Krste Asanovic for offering me the opportunity to be the teaching assistant of the 6.823 course in Fall 2006. This was a wonderful learning experience. For the first time in my life, I was responsible for giving tutorials, writing quizzes and exams. Before the course, I had never imagined that the effort needed in teaching a course was substantially greater than taking it as a student. This experience made me appreciate all my previous teachers and lecturers more.

This thesis is about the development of the Airblue platform, a project to which many students and post-docs from MIT have contributed. Many thanks to Kermin Elliott Fleming, Geoffrey Challen, Mythili Vutukuru, Samuel Gross, Quentin Smith, Hariharan Rahul and Ramakrishna Gummadi. I am especially thankful to Kermin Elliott Fleming. He is the co-leader of the Airblue project. Most of the infrastructure and tools for Airblue were built by him. Without him the project would not have gone this far. I am grateful to have Prof. Hari Balakrishnan as an advisor to this project. His expertise in computer networks helped us to understand what kinds of interesting problems in wireless research Airblue can help to solve. I would also like to thank Jamey Hicks, Gopal Raghavan and John Ankcorn from Nokia Research Center Cambridge (NRCC) for collaborating with us in this project and providing us the first version of the Airblue hardware.

As a graduate student, office was like my second home. Officemates were the people I spent most of my time with in the last few years. Thanks to my officemates Abhinav Agarwal, Asif Khan, Byungsub Kim, Chih-Chi Cheng, Jaewook Lee, Vinson Lee, Chun-Chieh Lin and Muralidaran Vijayaraghavan. Before moving to Stata Center, I shared the office with Byungsub and Vinson at Laboratory of Computer Science (LCS). They were the funniest officemates I have ever had. I remembered we always had mini dunk contests at our office. Byungsub and I joked that we were Malone and Stockton of LCS because we were going to spend a long time before graduating. It turned out that our prediction was spot on! Abhinav, Asif, Chih-Chi, Chun-Chieh, Jae and Muralidaran were my officemates at Stata Center. Abhinav and I also lived in the same apartment for the last 3 years. He taught me how to make a simple Indian dish, which became my popular dish. Asif brought us snacks and cakes whenever there was a big event from his family. I was certain he had a big family by the fact that we had free food from him way too often. Both Chun-Chieh and Chih-Chi were from Taiwan and passionate about high-end headphones. Thanks to them, I can now distinguish the sound quality difference between the low-end and high-end pairs. Jae was the nicest person I have ever met. He would never reject a request if he could fulfil it. He was also very organized. He filed all the documents properly. I doubt if he has ever lost anything. Murali was the "theorem proving machine" at our office. Whenever we had some ideas whose problems or solutions had not been well-defined, he would help us to formalize the problems and then prove for correctness.

I thank Nirav Dave, Myron King, Michael Pellauer, Daniel Rosenband and Richard Uhler who are my fellow group members in the Computation Structures Group (CSG). Although we mostly worked on different research projects, technical discussions with them about their projects always ended up with some insights into my own projects. I also enjoyed the occasions when we worked together to win several MEMOCODE design contests.

Being originally from Hong Kong, I have always been emotionally attached to my hometown. I was fortunate to make friends with other Hong Kong students in the area. My homesickness was cured whenever I watched Hong Kong movies or discussed

Hong Kong news and politics with them. Thanks to Christian Chan, Clement Tze Yan Chan, Vincent Chi Kwan Cheung, John Sie Yuen Lee and David Yue Tan Tang.

Throughout my Ph.D study, I did four industrial internships: two at IBM and two at Qualcomm. The experience equipped me with better teamwork and communication skills. Both my mentors, Dr. Suhas Pai from Qualcomm and Dr. Xiaowei Shen from IBM, treated me exceptionally well. Suhas brought me and another intern to swim once a week. He also invited us to his home for dinner at the end of the internship. His warm reception was one of the reasons that I decided to become his colleague after I graduate. Xiaowei drove me home several times when I missed the last shuttle to home. We were both supporters of the Chinese team in the 2004 Olympics.

Long distance relationship is always difficult to maintain. I was fortunate to be able to do so with my girlfriend, Edith Chiu. It was largely due to her being always supportive and considerate to me, especially during the stressful time when I was completing this thesis. The thought of chatting with her on phone after a long busy day motivated me to finish my work of the day as soon as possible. I am confident that we will be together for many years to come.

I consider myself to have a close relationship with my brother, Anthony. It was always fun to discuss with him about random topics and secrets between us. Each time after the conversation, I would be cheered up and ready to resume work.

Finally, I pay my highest tribute to my parents. Throughout my life, they have always been supportive to all my decisions, including those that they did not necessarily agree with. They were great teachers to me, giving me lessons in life beyond the classroom. I remember that it was my dream to study at MIT and become a scientist when I was young. When I told them my thought, they told me that I would need to work extremely hard to accomplish this goal. They were also my friends. I could share all my thoughts with them without any barrier. It is my greatest fortune to have them as my parents.

Contents

List of Figures

Chapter 1

Introduction

This thesis is about an innovative implementation of an FPGA-based wireless platform that is easily modifiable for wireless protocol development and experimentation.

Today's high-speed wireless networks are designed using the same layering principles, shown in Figure 1-1, that worked well for wired networks. In this design, the physical and link layers ensure a reasonably reliable point-to-point communication abstraction, on top of which higher layers implement routing, congestion control, and end-to-end reliability. The prevalent interfaces work well for wired networks because the physical characteristics of the transmission medium is relatively stable for a long period of time. As a result, most information about the channel conditions obtained from the physical layer is abstracted away from higher layers without severe performance impact.

Unfortunately, a radio is very different from a wire in two major aspects: first, it is a shared medium, so any two concurrent communications in the same frequency band will interfere with each other; second, radio signal received by a mobile node can suffer from various physical phenomena such as noise, multipath induced fading and Doppler shift. Both aspects make the channel characteristics change frequently and unpredictably. Therefore, an efficient wireless protocol must adapt promptly to such changing channel characteristics. This requires multiple layers of the network stack to share information about the channel conditions and act cooperatively to achieve higher throughput. A protocol that satisfies this requirement is considered to

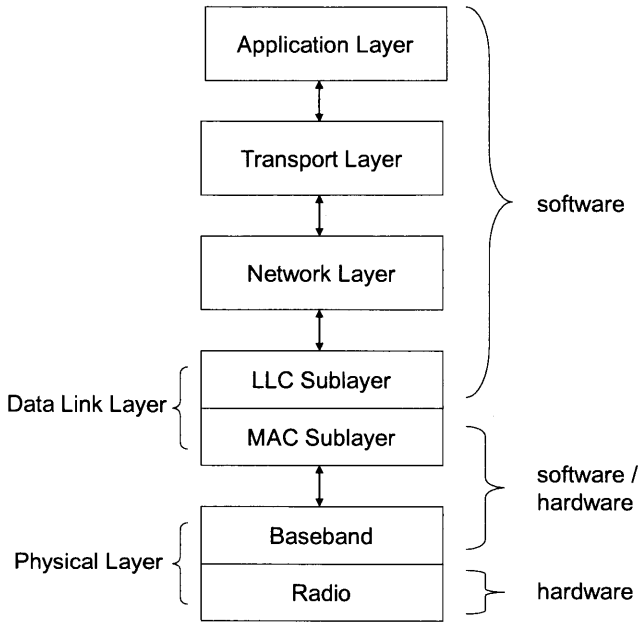


Figure 1-1: Overview of a wireless protocol stack

be a *cross-layer* wireless protocol. Over the past few years, researchers have proposed many cross-layer protocols. Some of the examples are interference cancellation [36], ZigZag decoding [30], Conflict Maps (CMAP) [92], the SoftPHY interface [45], SampleWidth [61], Analog Network Coding [49], MIXIT [50], COPE [51], VWID [33], ODS [32], SWIFT [73] and MIM [76]. All these proposals advocate some kind of modifications to the current layering structure by exposing various channel information obtained from the physical layer to higher layers.

1.1 The Case for a Prototyping Platform to Evaluate New Wireless Protocols

In order to evaluate and validate a cross-layer protocol, both simulations with channel models and real-time on-air experimentations are required. Simulations with channel models are useful to verify that the cross-layer protocol under evaluation is able to accurately estimate various channel conditions. Unfortunately, it is hard for any

channel model to model all the channel variations accurately because the interference in the channel itself is dependent on the protocol the communicating nodes use. The problem is exacerbated by high user densities. As a result, real-time on-air experimentations are also required to validate the benefits of the proposed protocol. It will be undesirable if two different sets of source code are written for simulations and on-air experimentations. Researchers need a unified platform that allows them to perform both efficiently. For such a platform to be useful, it needs to have the following four properties:

- **Configurability.** The platform must be flexible enough to implement various protocols. In addition, it is desirable for the platform to facilitate code reuse so that a new protocol can be implemented with minimal effort.
- **Observability.** To facilitate verification, the platform must provide designers easy means to isolate the source of the problem when a bug occurs. It is also important that the platform allows designers to easily add monitoring so that they can collect useful statistics from experiments.
- **Optimizability.** The platform must facilitate modular refinements so that designers can easily modify an individual block in the transceiver to meet the performance target without concerns about breaking the correctness of the whole transceiver.
- **Low-latency and high-throughput.** The platform must be capable of implementing protocols that require low-latency and high-throughput.

1.2 Problems of Existing Platforms

A variety of platforms have been proposed in the past few years to facilitate the development of new wireless protocols. These platforms span the entire design space of hardware and software implementations of the PHY and MAC.

1.2.1 Software-Based Radios

On one end of the spectrum are software-based platforms such as GNURadio [29], which provide the flexibility to modify any the layer of the stack. A pure software radio usually consists of an radio frequency (RF) front-end connected to a commodity PC through a peripheral IO interface, with all the processing, starting from the application layer down to the baseband, being done on the PC. This approach can easily satisfy the requirements of configurability and observability because of the ease of writing and debugging software programs. However, GNURadio is implemented on sequential C++ and does not scale easily to handle commercial throughputs. More importantly, GNURadio is not suitable for implementing low-latency protocols. Schmid *et al.* [78] show that the latencies introduced by the OS overhead as well as the IO communications between the RF and the PC are too large to meet the tight latency requirements of modern MAC protocols.

The SORA platform [86] combats the throughput problem by applying more sophisticated parallel programming techniques on multi-cores. For latency, SORA augments its software radio architecture with a more capable FPGA, which is used to store *pre-computed waveforms* of time-critical data such as the acknowledgment frames on the FPGA, thus avoiding the IO latency overhead suffered by GNURadio. As a result, SORA is able to achieve throughput and latency comparable to 802.11 hardware. This implementation is impressive, but is not convenient for developing new protocols for several reasons. First, the designer of a new protocol must manually re-partition the software blocks for multiple CPU cores and, assign the time-critical protocol portions to the FPGA. This has to be done even if the new protocol requires changes in only a few blocks. A recent paper reports that the PHY no longer met the timing when the authors changed the 802.11b PHY to perform more complicated decoding [85]. The only way to meet the timing was to parallelize the code further to exploit mult-cores. Second, it is impossible to achieve microsecond-latency communication between the MAC and the PHY, making it hard to implement several cross-layer mechanisms that require such communication (e.g., sending channel qual-

ity feedback from the PHY in the link-layer ACK). As an aside, SORA also has a form-factor concern: an 8-core PC is not so convenient for mobile experiments.

The latency problem of SORA or GNURadio stems from the fact that commodity PC architectures are optimized for throughput but not latency. Their IO specifications normally hide long latency by transferring data in large chunks. These limitations can be avoided by custom-designed System-on-Chips (SoCs) for configurable radios purposes. This is the approach taken by many commodity software radios like Vanu's software radio [88], Picochip's multi-core DSP [70], Intel's Scalable Communication Core (SCC) [39]. All these platforms employ a heterogeneous multi-core architecture consisting of flexible processing elements or accelerators that can efficiently execute the signal processing algorithms in common protocols. They also include Network on Chip (NoC) that can flexibly route data between computational resources. These custom systems are flexible and extensible for future protocols. However, they are not suitable tools for protocols designers because the programming model for these heterogeneous multi-core systems requires programmers to know the underlying architectures to make efficient use of available computational resources. This requires a steep learning curve for designers to program these systems. Moreover, these systems are usually developed without backward-compatibility in mind. Therefore, the knowledge obtained for one generation of the system may not be applicable to future generations. Finally, these systems usually cost at least ten-thousand dollars per node, making them unsuitable for academic research.

1.2.2 FPGA-Based Radios

To solve the latency problem, researchers have built radio platforms using Field Programmable Gate Array (FPGA) [67]. A FPGA chip contains programmable logic components that can be configured dynamically to execute a hardware design, which is usually described in Hardware Description Language (HDL) at Register Transfer Level (RTL). Although a design running on FPGA is typically slower than its Application-Specific Integrated Circuit (ASIC) counterpart, it usually outperforms the corresponding software implementation by two orders of magnitude because it is

easy for a design described in HDL to exploit fine-grain parallelism. For example, WiGLAN [24, 74] is a FPGA-based system that implements a high-throughput wide-band protocol achieving a data rate of multiple hundreds Mbps. This is beyond the capability of any current software radio.

One drawback of using FPGA for configurable radio is that describing an algorithm in conventional HDLs, like Verilog or VHDL, is much harder than describing the same algorithm in a software programming language. Although Verilog and VHDL support parameterized designs, the standard practice is to do such parameterization using a scripting language to generate specific RTL versions. Moreover, designs written in Verilog or VHDL are usually timing sensitive. This prevents the designs from being used as black boxes. These issues limit the configurability, observability and optimizability of the radio platform.

To make FPGA-based radio platforms more appealing to protocol designers, commercial vendors have developed tools [95, 1, 84] that can convert MATLAB or Simulink models to FPGA implementations. Basically, these vendors choose a large set of MATLAB or Simulink functions and then create hand-optimized hardware blocks for those functions. By doing so, the tools can instantiate the appropriate block when a protocol designer specifies to use a library function in MATLAB or Simulink. However, if a designer invents an algorithm that is unsupported, he will need to implement the design in HDL. To make the matter worse, the RTL generated by these tools is hard to modify because it lacks modularity. Rice's WARP [93] is a platform developed using this approach. Its baseband PHY is implemented with Verilog blocks generated by Xilinx System Generator. As a result, implementing protocols that require PHY modifications can only be carried out by experienced users with a deep understanding of the whole design.

Taking one step further, AccelFPGA compiler [5] can synthesize user created functions in MATLAB to RTL, which can later be synthesized onto FPGA by FPGA synthesis tools. AccelFPGA provides programmers directives to control the resource usages of the output designs, thus permitting programmers to use these directives for design space exploration. From our experience (see Chapter 6), these kind of

Parameter	Value
Carrier Frequency	2.4 GHz
Bandwidth	20 MHz
Number of Data Subcarriers	48
Number of Pilot Subcarriers	4
Number of Unused Subcarriers	12
FFT/IFFT Period	3.2 μs
Cyclic Prefix (CP)	0.8 μs
Symbol Period	4 μs
Modulation per Bin	BPSK, QPSK, 16 & 64 QAM

Figure 1-2: Summary of Airblue's 802.11g Transceiver Specification

tools, which generate hardware designs from some sequential software languages, only perform well on blocks with regular structures and simple controls. This is because it is difficult to express fine-grain parallelism of an irregularly-structured algorithm in a sequential language. We find that the best designs are usually achieved by using these tools to generate small blocks and then write glue logic connecting the blocks in conventional HDL. However, this approach increases the verification effort because it forces designs to be developed in multiple languages.

1.3 Contributions

In this thesis, we present an alternative method to implement configurable radios on FPGA: We describe all the hardware blocks of a radio transceiver in a HDL language called Bluespec. In Bluespec, a programmer describes the execution semantics of the design through Term Rewriting Systems (TRS). Then, the Bluespec compiler converts the TRS into RTL designs. Hoe *et al.* [38] show that efficient hardware designs can be generated from TRS. Different from an interface in Verilog or VHDL, which is a collection of wires with no semantic meaning, the Bluespec interface automatically includes necessary handshake signals for communication between blocks. Therefore, Bluespec facilitates latency insensitive designs which are essential to system construction via modular composition. The main contributions of this thesis are:

- **An innovative approach to implement a configurable radio on an**

FPGA. We propose that the design of a configurable radio must have two design properties, namely *latency-insensitivity* and *data-driven control*, to facilitate modular refinements. We propose the design principles to obtain designs with the required properties. Following these principles, we developed Airblue, an 802.11g compliant radio implemented on an FPGA-based wireless platform. Figure 1-2 summarizes the Airblue's 802.11g transceiver specification. In addition to the flexibility, Airblue meets the tight latency requirements by implementing both the PHY and the MAC on FPGA and connecting them with *streaming interfaces*, which allow data to be processed as soon as it is ready. Finally, we show that we can easily extend Airblue to become a high-speed FPGA-accelerated simulator by replacing the RF devices with software channels through *virtualization of FPGA platforms*. Again, this is only possible because of the latency-insensitivity of our designs.

- **Implementations of highly reusable designs through parameterizations.** We discuss the difference of two kind of parameterizations: compile-time and run-time and show how we can leverage the high-level language features provided by Bluespec, such as type checking and polymorphism, to implement reusable designs in each kind of parameterization. We show that different combinations of the two kinds of parameterizations can be used to generate different design points on the algorithmic flexibility versus hardware complexity tradeoff curve. In addition, we present several techniques which allow designers to reuse existing designs to implement run-time parameterized designs.

- **Protocol experiments using Airblue.** We show the usefulness of Airblue by implementing SoftRate [91], a cross-layer bit rate adaptation protocols that can improve throughput. Through this example, we show why the support of both simulations and on-air experimentations in a unified framework like Airblue is important to validate a protocol.

- **Comparison of high-level design methodologies.** We compare the two competing high-level design methodologies, i.e., Bluespec and C-based synthe-

sis, for hardware implementations of digital signal processing (DSP) algorithms. We show that C-based synthesis tools are more effective than Bluespec in early stages of the design development but Bluespec produces much better final hardware. We argue that C-based synthesis will be of limited use in final FPGA implementations, especially for high performance blocks required by future wireless protocols, thus justifying Bluespec as the language of choice for Airblue.

1.4 Thesis Organization

We begin with the discussion of implementing a generic transceiver based on the Orthogonal Frequency-Division Multiplexing (OFDM) modulation scheme in Chapter 2. Chapter 3 presents the implementation of Airblue. Chapter 4 shows the flexibility of Airblue by implementing SoftRate [91]. Chapter 5 discusses the techniques of making Airblue more flexible by reusing existing designs to implement run-time parameterized designs. Chapter 6 compares the two competing high-level design methodologies, i.e., Bluespec and C-based synthesis, for hardware implementations of digital signal processing (DSP) algorithms. Chapter 7 summarizes thesis and discusses possible future work.

Chapter 2

A Generic OFDM Transceiver

Orthogonal Frequency-Division Multiplexing (OFDM) has become the preferred modulation scheme for both broadband and high bit rate digital wireless protocols because of its spectral efficiency and robustness to multipath interference. Although the components and overall structure of different OFDM protocols are functionally similar, the characteristics of the environment for which a wireless protocol is designed often result in different instantiations of various components. In this chapter, we describe how we can instantiate baseband processoring of two different wireless protocols, namely 802.11a and 802.16 in Bluespec from a highly parameterized code for a generic OFDM protocol. Our approach results in highly reusable IP blocks that can dramatically reduce the time-to-market of new OFDM protocols. Using a Viterbi decoder we also demonstrate how parameterization can be used to study area-performance tradeoff in the implementation of a module. Furthermore, parameterized modules and modular composition can facilitate implementation-grounded algorithmic exploration in the design of new protocols.

The content of this chapter is revised from a publication presented at the 2007 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) [63], of which I am the first author. *This research is funded by Nokia Inc. (2006 Grant to CSAIL-MIT) and the National Science Foundation (Grant #CCF-0541164).*

2.1 Introduction

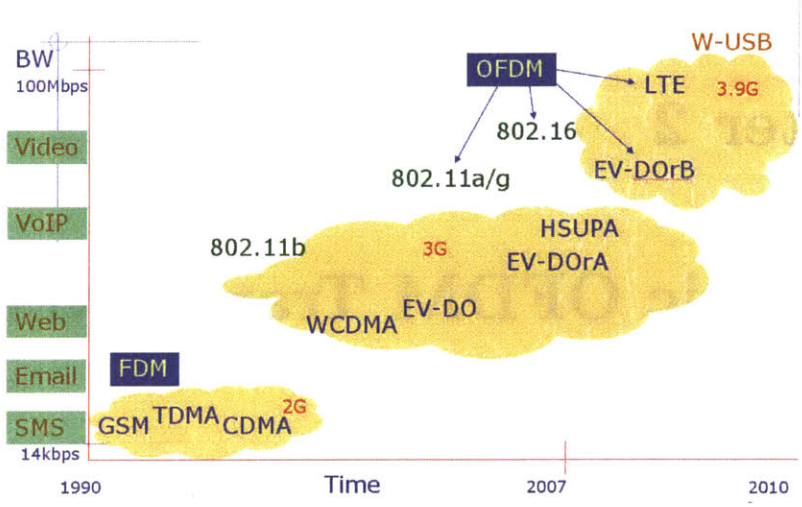


Figure 2-1: Digital wireless standard evolution

Since the early 1990's, there has been a rapid evolution in digital wireless protocols to enable higher data rates, improve bandwidth efficiency and to offer services to more users. There is a dramatic shift from purely voice based services to high bit rate data services to support web browsing, VoIP and high definition video. Another interesting development is the convergence of services offered via broadband wireless access like WiMAX and cellular networks like 3G/4G. The underlying technology that enables this high data rate in non-line-of-sight environment is a modulation scheme known as Orthogonal Frequency Division Multiplexing (OFDM) [17]. OFDM has been around for several decades, but now its robustness to multipath interference has been proven in practice by the widespread deployment of 802.11a/g and ADSL.

Some of the challenges in wireless communication are interference from other RF sources, self-interference due to multipath transmissions, and frequency dependent signal loss (fading). In the narrowband environment, simple modulation schemes, such as frequency modulation (FM), amplitude modulation (AM), and phase modulation (PM), protect against interferences and signal loss. However, such simple schemes do not offer high data transmission capacity. With higher channel band-

width and greater rates of mobility, inter and intra-symbol interference for a single carrier becomes significant. Traditional FDM (Frequency Domain Modulation) techniques have larger guard bands between subcarriers and waste bandwidth. OFDM offers an elegant solution by spreading data across many closely-packed and overlapping narrowband subcarriers. In OFDM, the spacing between sub-carriers is carefully designed such that they are orthogonal to each other, meaning the product of two carriers with different frequencies is zero if sampled at frequency determined by sub-carrier spacing. This results in zero cross-talk between sub-carriers. In short, to get high data rates we need higher bandwidth. To avoid self-symbol interference and inter symbol interference the bandwidth is divided into multiple narrowbands that carry lower data rates and the sub-carriers are placed orthogonal to each other. As a result, OFDM provides high spectral efficiency and is robust against multipath interferences.

Modern cell phones usually contain multiple radios, most of which are "OFDM based" and are implemented as special hardware blocks. Therefore, it would be beneficial to share the design cost of such radios by building a set of highly reusable IP blocks, which can be instantiated with different parameters for different protocols. While the development of such modules requires domain expertise, we think the use of such modules in designing and implementing new protocols requires considerably less knowledge.

In this chapter, we demonstrate that it is possible to generate efficient hardware for two different wireless protocols, namely 802.11a [43] and 802.16 [44], from the same code base written in Bluespec SystemVerilog (BSV). The following two features are essential for such designs: 1) the ability to compose independently created modules with predictable functionality and performance and 2) a polymorphic type system that permits highly parameterized codes. The first feature permits the refinement of individual modules to meet the performance objectives and exploration of area-performance tradeoffs without exacerbating the verification problem. The latter capability allows many of such designs to be described in a concise source code. As a further evidence of the flexibility and reusability of our design, Pitkanen *et al.* [71] were able to build a 802.15.3 (WUSB) transmitter using our OFDM framework in six

weeks with two designers.

Although Verilog and VHDL support parameterized designs, the standard practice is to do such parameterization using scripts written in a language like Perl that generate specific RTL versions. It may be possible to do highly parameterized designs in SystemC, though we have not seen it reported in the literature. SystemC, unlike Bluespec, only provides limited capability to synthesize parameterized designs into efficient hardware. Parameterization in Bluespec, on the other hand, can be used freely because it does not cause any extra logic gates to be generated; the compiler removes all the static parameterization during the "static elaboration" phase.

The remaining of the chapter is organized as follows: We begin by describing the processing blocks in a generic OFDM baseband transceiver in Section 2.2. We also explain how the various blocks of 802.11a and 802.16 protocols are instances of these generic blocks. Next, we show the overall structure of the OFDM transceiver in Bluespec and discuss the issues related to parameterization in Section 2.3. We further discuss the parameterization of modules, using the example of Viterbi decoder, for architectural exploration to meet area, power and performance goals in Section 2.4. After that, we describe the implementations of several computationally intensive blocks in the transceiver. Finally, we present synthesis results in Section 2.6, related work in Section 2.7 and our conclusions in Section 2.8.

2.2 Generic OFDM Baseband Transceiver

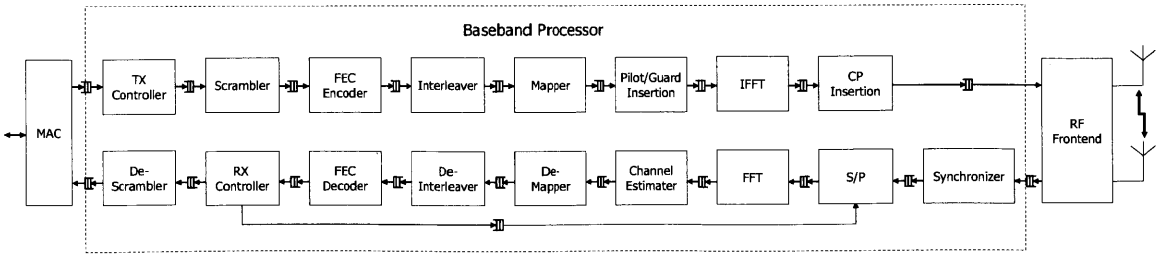


Figure 2-2: Block digram of a generic OFDM baseband transceiver

The structure of a generic OFDM baseband transceiver is shown in Figure 2-2. In wireless communications, the fundamental unit of communication is a *symbol*, which encodes one or more data bits. An OFDM symbol in turn is defined as a collection of digital samples, which are usually represented as complex numbers. The size of the symbol is determined by the number of subcarriers used in the system. In general, a fraction of subcarriers, known as data subcarriers, are used for data transmission and the remaining subcarriers are used for pilots and guard bands. Pilots provide information which is used by the receiver to estimate the frequency fading in transmitted symbol better, thereby increasing the chance of a successful data reception. Guard bands are normally added to both sides of the frequency spectrum to avoid interference with other carriers.

The data received from the Medium Access Control (MAC) layer flows through various processing steps, which convert it to an OFDM symbol that is transmitted over the air. Similarly, on the receiver side, OFDM symbols are formed from the signals received through the A/D converter and processed through various stages. Finally, the resulting received data is sent to the MAC. In the rest of this section, we briefly describe each of the blocks in these transceiver pipelines, and discuss parameters for each block that are needed to describe two specific OFDM-based protocols, namely 802.11a and 802.16. In each case, we also point out the performance required to meet the standard where it is appropriate. The table in Figure 2-3 summarizes the parameters used by various blocks in both protocols. It should be read in conjunction with the following description of the transceiver pipelines.

2.2.1 Transmitter Pipeline

TX Controller: Receives information from the MAC and generates the control and data for all the subsequent blocks.

Scrambler: Randomizes the data bit stream to remove repeated patterns, like long sequences of zeros and ones. This enables better results for Forward Error Correction (FEC). A scrambler is usually implemented with linear feedback shift registers (LFSR). An LFSR has two algorithmic settings: the size of the shift register and the

	802.11a	802.16
Scrambler		
shift register size	7 bits	15 bits
linear function	$x^7 + x^4 + 1$	$x^{15} + x^{14} + 1$
throughput	54Mbps	26.2Mbps
FEC Encoder (Reed-Solomon)		
encoder profile (N,K,T)	NA	(255,239,8)
supported profiles (N,K,T)	NA	(12,12,0), (32,24,4), (40,36,2), (64,48,8), (80,72,4), (108,96,6), (120,108,6)
throughput	NA	29.1Mbps
FEC Encoder (Conv)		
constraint length	7	7
supported rates	1/2, 2/3, 3/4	1/2, 2/3, 3/4, 5/6
generator polynomials	133_{OCT} & 171_{OCT}	171_{OCT} & 133_{OCT}
throughput	72Mbps	35Mbps
Interleaver		
block size (bits) (blockSize)	48, 96, 192, 288	192, 384, 768, 1152
throughput	1 block per $4\mu s$	1 block per $33\mu s$
Mapper		
modulations	BPSK, QPSK, 16-QAM, 64-QAM	BPSK, QPSK, 16-QAM, 64-QAM
throughput	48 samples per $4\mu s$	192 samples per $33\mu s$
Pilot/Guard Insertion		
pilot indices	-21, -7, 7, 21	-88, -63, -38, -13, 13, 38, 63, 88
guard indices	-32 to -27, 0, 27 to 31	-128 to -101, 0, 101 to 127
throughput	64 samples per $4\mu s$	256 samples per $33\mu s$
IFFT		
size	64	256
throughput	64 samples per $4\mu s$	256 samples per $33\mu s$
CP. Insertion		
CP size	16 (32 for preamble)	8, 16, 32, 64
short preamble	8 16-sample symbols	4 64-sample symbols
long preamble	2 64-sample symbols	2 128-sample symbols
throughput	80 samples per $4\mu s$	264 samples per $33\mu s$
Synchronizer		
preamble settings	same as CP. Insertion	same as CP. Insertion
throughput	20M samples per sec	8M samples per sec
S/P		
symbol / CP sizes	64 / 16	256 / 8, 16, 32, 64
throughput	64 samples per $4\mu s$	256 samples per $33\mu s$
FFT		
size	64	256
throughput	64 samples per $4\mu s$	256 samples per $33\mu s$
Channel Estimator		
preamble settings	same as CP. Insertion	same as CP. Insertion
pilot/guard settings	same as Pilot/Guard Insertion	same as Pilot/Guard Insertion
throughput	48 samples per $4\mu s$	192 samples per $33\mu s$
Demapper		
demodulations	BPSK, QPSK, 16-QAM, 64-QAM	BPSK, QPSK, 16-QAM, 64-QAM
throughput	288 decisions per $4\mu s$	1152 decisions per $33\mu s$
Deinterleaver		
block size (decisions)	48, 96, 192, 288	192, 384, 768, 1152
throughput	1 block per $4\mu s$	1 block per $33\mu s$
FEC Decoder (Viterbi)		
conv. code settings	same as Conv. Encoder	same as Conv. Encoder
throughput	54Mbps	29.1Mbps
FEC Decoder (Reed-Solomon)		
Reed-Solomon settings	NA	same as Reed-Solomon Encoder
throughput	NA	26.2Mbps
Descrambler		
LFSR settings	same as Scrambler	same as Scrambler
throughput	54Mbps	26.2Mbps

Figure 2-3: Algorithmic settings of 802.11a and 802.16 transceivers

linear function for generating the feedback.

In 802.11a, the size of the shift register is 7 bits and the linear function is x^7+x^4+1, while in 802.16, the size of the shift register is 15 bits and the linear function is $x^{15} + x^{14} + 1$.

FEC Encoder: Encodes data and adds redundancy to the bit stream to enable the receiver to detect and correct errors.

Both protocols use convolutional coding with constraint length 7 at rate 1/2. The generator polynomials are G1 = (1,0,1,1,0,1,1) and G2 = (1,1,1,1,0,0,1).

802.16 also requires Reed-Solomon encoding before the data is passed to the convolutional encoder. Reed-Solomon encoding itself has several parameters, for example, a (N=255, K=239, T=8) encoder takes 239 bytes of data and adds 2×8 bytes of parity to produce a 255 byte coded message. As shown in Figure 2-3, support for several setting of such parameters are required by the 802.16 standard.

Both protocols also use a technique known as *puncturing* to reduce the transmitted number of bits. For higher transmission rates in low-noise channels, the encoded data is *punctured* by deleting bits before transmission and replacing them with fixed values on reception. This reduces the number of bits to be carried over the channel and depends on the decoder to correctly reconstruct the data.

Interleaver: Rearranges blocks of data bits by mapping adjacent coded bits into non-adjacent subcarriers to protect against burst errors. The block size is the same as the number of bits that are coded in a single OFDM symbol. The symbol size itself is determined by the number of data subcarriers and the modulation scheme employed.

802.11a uses 48 data subcarriers while 802.16 uses 192 data subcarriers. Both protocols support BPSK, QPSK, 16-QAM and 64-QAM modulation schemes.

Mapper: Passes interleaved data through a serial to parallel converter, mapping groups of bits to separate carriers, and encoding each bit group by frequency, amplitude, and phase. The output of the Mapper contains only the values of data subcarriers for an OFDM symbol.

Pilot/Guard Insertion: Adds the values for pilot and guard subcarriers. The

subcarrier indices are protocol-specific. Both protocols use scramblers to generate values for the pilots and use null values for the guard subcarriers.

The algorithmic settings of the pilot scrambler in 802.11a are the same as the settings it uses for the data scrambler. On the other hand, the pilot scrambler for 802.16 uses the linear equation $x^{11} + x^9 + 1$.

IFFT: Converts symbols from the frequency domain to the time domain. The size of the IFFT is determined by the number of subcarriers used by the given OFDM protocol.

The IFFT size for 802.11a is 64 points and fixed , while for 802.16, the IFFT size is scalable from 128 points to 2048 points.

CP Insertion: Copies some samples from the end of the symbol to the front to add some redundancy to the symbols. These duplicated samples are known as a cyclic prefix (CP). The purpose of the cyclic prefix is to avoid Inter-Symbol Interference (ISI) caused by multipath propagation.

For 802.11a, the size of CP is 16 (1/4 of a symbol) and fixed, while in 802.16, the size of CP varies from 1/32 to 1/4 of a symbol.

This block also adds a preamble before the first transmitted symbol. A preamble is a collection of predefined complex numbers known by the receiver so that it can detect the start of new transmission. The preambles for the two protocols have similar structure.

For 802.11a, S and L are 16 and 64 samples, respectively, and S is repeated 8 times in the short preamble. For 802.16, S and L are 64 and 128 samples, respectively, and S is repeated 4 times in the short preamble. The CP size is 32 for 802.11a and either 8, 16, 32 or 64 for 802.16.

After CP insertion, the symbol are converted into analog signals by D/A converter and transmitted through the air.

2.2.2 Receiver Pipeline

The functionality of the blocks in the receiver is roughly the reverse of the functionality of their corresponding blocks in the transmitter. However, since the receiver has to

recover data from a degraded signal, some receiver blocks have to do more processing and consequently require more implementation effort. When the antenna detects the signal, it amplifies the signal and passes it to the A/D converter to generate baseband digital samples.

Synchronizer: Detects the starting position of an incoming packet based on preambles. It is extremely important for the synchronizer to correctly estimate the OFDM symbol boundaries so that subsequent blocks process appropriate collection of samples together. In many implementations, the synchronizer also detects and corrects carrier frequency offset that is caused by the difference in the oscillator frequencies at transmitter and receiver or due to the Doppler Effect. The synchronizer uses the preamble to perform timing and frequency synchronization. There are many different implementations of the synchronizer, most of which involve auto-correlation and cross-correlation. For the synchronizer to support different protocols, it needs to know the preamble structure, the symbol size and the CP size of the protocol.

Serial to Parallel (S/P): Removes the cyclic prefix (CP) and then aggregates samples into symbols before passing them to the FFT. It also propagates the control information from the RX Controller in order to configure subsequent blocks.

FFT: Converts OFDM symbols from the time domain back into the frequency domain.

Channel Estimator: Uses the information from pilots to estimate and compensate for frequency-dependent signal degradation. The channel estimator estimates and corrects the errors caused by multipath interference. As in the case of the synchronizer, there are many different algorithms for channel estimation. Many of them use either the preambles or the pilots to estimate the effect of the interference on each data subcarrier. We parameterize the channel estimator by protocol-specific preamble and pilot values.

Demapper: Demodulates data and converts samples to encoded bits, which are used by the FEC decoder. The number of encoded bits generated per sample is determined by the specific modulation scheme. The parameters of this block are modulation schemes supported and the functions for converting samples to decisions.

Deinterleaver: Reverses the interleaving performed by transmitter and restores the original arrangement of bits.

FEC Decoder: Uses the redundant information that was introduced at the transmitter to detect and correct any errors that may have occurred during transmission. Both 802.11a and 802.16 use the Viterbi algorithm [89] to decode convolutionally encoded data. To support multiple protocols, the decoder uses the same parameter settings as the convolutional encoder at the transmitter side. Since 802.16 also uses Reed-Solomon encoding, corresponding Reed-Solomon decoder that supports appropriate profiles is used in the receiver side.

Descrambler: Reverses the scrambling performed by the transmitter.

RX Controller: Based on the decoded data received from Descrambler, the RX Controller generates the control feedback to S/P block.

2.3 General Considerations for Parametric Implementations

We employ several techniques to enable significant module reuse and customization across different protocols, architectures, and design points. The high-level structure in which modules are interconnected follows a transaction-level modeling style. Furthermore we restrict communication between the modules to pass messages containing control and data values. The control part is not modified by any module as the message flows through the pipeline and is stripped off before the message leaves the baseband processing section. The control part varies in type and value for different protocols; a challenge in designing reusable modules is to relate different (dynamic) control information to different (static) instantiations of a module. In this section, we illustrate this point using the parameterized coding of the Scrambler block.

2.3.1 Transaction-Level Modeling Style Interfaces

In order to decouple modules, the interface of each module is implemented with the ready/enable handshaking approach, which is embodied in the `Put` and `Get` interfaces [20]. The Bluespec compiler automatically enforces ready/enable handshaking between modules connected in this manner such that an upstream module will block if the downstream module is not ready. This interface style is compatible with transaction-level modeling (TLM) approach [31]. The interface of each module is declared as follows:

```
interface Block#(type in_mesg, type out_mesg);
   interface Put#(in_mesg)  in;
   interface Get#(out_mesg) out;
endinterface
```

This is a highly polymorphic definition in that the types of the messages going in and out of the module `Block` are themselves passed in as static parameters. Note that parameterized types are indicated with a hash mark (e.g. `Block#`).

The code above defines that the interface has an input method called `in` and an output method called `out` and these methods have `Put` and `Get` interfaces, respectively. `Put` and `Get` interfaces are part of the Bluespec library. By making use of the `mkConnection` function one can easily connect the `Get` method of a module to the `Put` method of another module provided the declared types match. The BSV compiler automatically generates the logic needed to transfer the data from the `Get` to the `Put` whenever both methods are ready.

The generic OFDM transmitter pipeline can be described using the `mkConnection` function:

```
mkConnection(tx_controller.out, scrambler.in);
mkConnection(scrambler.out, encoder.in);
mkConnection(encoder.out, interleaver.in);
mkConnection(interleaver.out, mapper.in);
mkConnection(mapper.out, pilotInsert.in);
mkConnection(pilotInsert.out, ifft.in);
mkConnection(ifft.out, cpInsert.in);
```

```
tx_controller <- mkProtocol_Controller(); // protocol-specific
scrambler     <- mkScrambler(scramblerCtrl, lfsrSz, lFunc);
encoder       <- mkEncoder(); // protocol-specific
interleaver   <- mkInterleaver(intrlvrCtrl, intrlvrGetIdx, blockSize);
mapper        <- mkMapper(mapperCtrl, invertInput);
pilot_insert  <- mkPilot_Insert(guardPos, pilotPos, pilotFuncs);
ifft          <- mkFFT_IFFT(ifftCtrl, ifftSize);
cp_insert     <- mkCP_Insert(cpCtrl, symbolSize);
```

Figure 2-4: Instantations of transmitter modules

```
module mkEncoder (Encoder80211);
    // state elements
    encoder_conv  <- mkConv_Encoder(poly_g_0, poly_g_1);
    encoder_punc  <- mkPuncturer(puncCtrl, puncFuncs);

    // connections
    mkConnection(encoder_conv.out, encoder_punc.in);

    // Get, Put methods
    interface in = encoder_conv.in;
    interface out = encoder_punc.out;
endmodule
```

Figure 2-5: 802.11a encoder

The modules of the receiver pipeline are connected in a similar fashion. Since the structure of the OFDM transceiver is the same across protocols, this portion of the code will remain the same regardless of the protocol we are implementing. The changes will appear when we instantiate the modules from the module definitions. BSV uses the symbol <- for module instantiation. In Figure 2-4 , we show how one can instantiate the modules for the transceiver using module definitions (the names starting with with `mk`). All module definitions are generic except those that instantiate the controllers, the encoder and the decoder:

Different OFDM protocols use different collection of FEC schemes. For instance, as shown in Figure 2-3, 802.11a uses convolutional encoder with puncture while and 802.16 uses Reed-Solomon encoder followed by convolutional encoder with puncture. The encoders are sufficiently different that sharing a parameterized module definition would be cumbersome. As a result, we made separate definitions, as shown in Figure 2-5 and Figure 2-6.

```
module mkEncoder (Encoder80216);
    // state elements
    encoder_rs    <- mkRS_Encoder(rs_encoderCtrl);
    encoder_conv  <- mkConv_Encoder(poly_g_0, poly_g_1);
    encoder_punc  <- mkPuncturer(puncCtrl, puncFuncs);

    // connections
    mkConnection(encoder_rs.out, encoder_conv.in);
    mkConnection(encoder_conv.out, encoder_punc.in);

    // Get, Put methods
    interface in = encoder_rs.in;
    interface out = encoder_punc.out;
endmodule
```

Figure 2-6: 802.16 encoder

OFDM Messages: In our OFDM library, blocks communicate with each other by passing OFDM messages, which consist of two fields: `control` and `data`. The type and format of the data is independent of the protocol being implemented, but the `control` encoding is protocol-specific and generated by the TX Controller or RX Controller. Consequently, we define the generic type for OFDM messages as shown in Figure 2-7.

```
typedef struct{
    ctrl_t                  control;
    Vector#(sz, data_t)     data;
} Mesg#(type ctrl_t, numeric type sz, type data_t);
```

Figure 2-7: Type definition of a generic OFDM message

The type of `control` is defined by `ctrl_t` and the type of `data` by a vector of `data_t` of size `sz`. `ctrl_t`, `data_t` and `sz` are static parameters which are evaluated at compiled time. Thus, the `in_mesg` and `out_msg` types are all instances of `Mesg` type. For example, the type of the message for the 802.11a Scrambler (`ScramMesg`) may be defined as follows, where types `ctrl_t` and `data_t` are instantiated to `Ctrl80211` and `Bit#(1)`, respectively:

```
typedef Mesg#(Ctrl80211, sz, Bit#(1))  ScramMesg#(numeric type sz);
```

Note that the `ScramMesg` type still has `sz` as a type parameter.

2.3.2 Parameterization of the Scrambler

In order to support various protocols, it is important that our implementations be as flexible as possible. We achieve this by parameterizing the implementation for both data types and widths. We illustrate this by describing the implementation of the scrambler module.

The scrambler randomizes the input bit stream by XORing each bit with a pseudo-random binary sequence (PRBS) generated by a linear feedback shift register (LFSR). For example, for the linear function $x^7 + x^4 + 1$, we first compute the `feedback` bit by XORing the 4th and 7th bit of the random number in `lfsr`. Then, we generate the output bit by XORing `inData` with the `feedback` bit. Finally, we compute the new random number by shifting the `feedback` bit into the current random number. For higher performance, we can process up to `steps` bits of `inData` at a time.

Figure 2-8 shows the function `scramble` which implements the LFSR. In BSV, functions are compiled to combinatorial circuits. This function takes as input the coefficients of the LFSR polynomial as integer vector `lFunc`, the initial value of the shift register `lfsr`, and the input bit vector `inData`. It returns a 2-tuple of values: the new value of the shift register and the scrambled data.

One important note about this code is that it represents a combinational circuit. All the `for` loops are completely unrolled at compile time to generate a DAG during the static elaboration phase of the BSV compiler. Consequently, the above definition can be compiled only if `lFunc` (the linear function), `lfsrSz` (the shift register size) and `steps` (the number of bits to be processed) are known at compile time, and they must be passed as parameters to the `mkScrambler` when it is used to instantiate a scrambler module.

With our definition, the `scramble` function can be used by both the 802.11a and the 802.16 protocols. When we use it in 802.11a, the `lFunc` will be a vector containing values 4 and 7, and the `fsz` will be 7. On the other hand, `lFunc` will contain 14 and 15 and `fsz` will be 15 when it is used in 802.16. The number of bits to be processed (`steps`) is not specified by the protocol and can be set to meet the performance goals.

```
function Tuple2#(Bit#(lfsrSz),Vector#(steps, Bit#(1)))
         scramble(Vector#(fsz, Integer)    lFunc,
                  Bit#(lfsrSz)             lfsr,
                  Vector#(steps, Bit#(1)) inData);

   Vector#(steps, Bit#(1)) outData;
   Bit#(lfsrSz) nextLfsr = lfsr;
   Bit#(1) feedback;
   for(i = 0; i < steps; i = i + 1)
   begin
      feedback = 0;
      // loop to generate the feedback bit
      for(j = 0; j < fsz; j = j + 1)
         feedback = nextLfsr[lFunc[j]] ^ feedback; // XOR
      // XOR feedback and inData for outData
      outData[i] = inData[i] ^ feedback;
      // shift feedback into LSB
      nextLfsr = {nextLfsr[n-2:0], feedback};
   end
   return tuple2(nextLfsr, outData);
endfunction
```

Figure 2-8: Scramble function

It turns out that the scrambler is not the slowest module and thus, even the `steps` value of one is sufficient to meet the performance requirements.

Dynamic Parameters: The scrambler can have three operational modes:

1. `Normal`: Input is randomized using the current LFSR state
2. `Bypass`: Input is forwarded without processing
3. `NewSeed`: The LFSR state is reset with a given value and then the input is randomized.

The information regarding how the input data is to be processed is specified in the control part of the message. This information is extracted by the `scramblerCtrl` function in the scrambler module rule as shown in Figure 2-9.

A challenge arises because different protocols use the scrambler differently. For example, the 802.11a protocol requires the scrambler not to scramble the header, while the 802.16 protocol requires the header to be scrambled too. This is why, as we pointed out earlier, the control part encoding is protocol-specific.

```
rule scrambling(True); //scrambling rule
   let mesg = inQ.first();
   let gCtrl = mesg.ctrl;
   let sCtrl = scramblerCtrl(gCtrl);
   let data = mesg.data;
   case (sCtrl) matches
      tagged Bypass: outQ.enq(Mesg{ctrl: gCtrl, data: data});
      tagged Normal:
         begin
            match {.oBits, .oSeed} = scramble(data, lfsr);
            lfsr <= oSeed;
            outQ.enq(Mesg{ctrl: gCtrl, data: oBits);
         end
      tagged NewSeed .nSeed:
         begin
            match {.oBits, .oSeed} = scramble(data, nSeed);
            lfsr <= oSeed;
            outQ.enq(Mesg{ctrl: gCtrl, data: oBits);
         end
   endcase
endrule
```

Figure 2-9: The scrambling rule that can operate at three different modes according to the information extracted by the scramblerCtrl function

```
function ScramblerCtrl scramblerCtrl80211(Ctrl80211 ctrl);
   return case (ctrl.region)
             Header:    Bypass;
             FirstData: NewSeed 7b101101;
             Data: Normal;
             Tail: Bypass;
          endcase;
endfunction
```

Figure 2-10: 802.11a scramblerCtrl function

We solve this problem by passing the `scrambleCtrl` function as a parameter to the scrambler module. In our implementation, the control encoding of both 802.11a and 802.16 contain a field called `region`. The possible values for the `region` field for 802.11a are `Header`, `FirstData`, `Data` and `Tail`; The possible values for the `region` field for 802.16 are `FirstData`, `Data` and `Tail`. In addition to the region field, the 802.16 control encoding also contains a field `seed` which specifies the value for the scrambler seed. Figure 2-10 and Figure 2-11 show the definitions of the `scramblerCtrl` for the two protocols respectively.

Figure 2-12 shows the definition of the `mkScrambler` with all the required param-

```
function ScramblerCtrl scramblerCtrl80216(Ctrl80216 ctrl);
   return case (ctrl.region)
              FirstData: NewSeed ctrl.seed;
              Data: Normal;
              Tail: Bypass;
          endcase;
endfunction
```

Figure 2-11: 802.16 scramblerCtrl function

```
module mkScrambler#(function ScramblerCtrl scramblerCtrl(ctrl_t ctrl),
                    Integer lfsrSz,
                    Vector#(fsz, Integer) lFunc)
       (Scrambler#(ctrl_t, steps));

    // state elements
    Reg#(Bit#(lfsrSz)) lfsr <- mkRegU;
    ... fifos (inQ, outQ) ...
    // Scrambling rule
    ...
    // Get, Put methods
    ...
endmodule
```

Figure 2-12: Module definition of mkScrambler

eters we discussed earlier.

2.4 Performance Tuning through Architectural Exploration

We explored various design alternatives in order to ensure that our design meets the performance goals. To facilitate such architectural exploration, we parameterize the model so that different designs can be instantiated from the same code base during the static elaboration phase. A parameterized FFT which can be instantiated with different microarchitecture at synthesis time is shown in [22]. In this section, we illustrate a way of parameterizing components of the Viterbi decoder to enable architectural explorations.

A Viterbi decoder uses the Viterbi algorithm [89] to decode bitstreams encoded using a convolutional forward error correction code. The algorithm determines the

most likely input bitstream given the received noisy encoded stream.

To understand the Viterbi algorithm, it helps to understand the convolutional encoder. A k-bit convolutional encoder is a state machine consisting of 2^k states, with transitions between states conditional on input bits and emitting a fixed number of output bits. In these two protocols, 2 bits are emitted per input bit. The current state of the encoder is named by the last k input bits. Because transitions are conditional on a single bit, each state of the encoder can be reached only from two previous states.

The Viterbi algorithm uses dynamic programming to find the most likely state transition sequence followed by the encoder given a received bit sequence. The algorithm retraces this sequence of states to reconstruct the original bit stream.

Too much time and memory would be required for the decoder to wait until it has received the entire data sequence before producing a result. However, we can achieve almost the same level of accuracy by recording only the last n transitions, and emitting one bit per timestep. In practice, a value of $n = 5(k + 1)$ yields satisfactory results. For 802.11a and 802.16, $k = 6$, so $n = 35$.

In our implementation, the Viterbi decoder consists of two modules: the path metric unit and the Traceback unit. The path metric unit contains a 2^k word memory, where each entry is essentially the probability that a sequence of input bits ended in that state. In practice, cumulative error between the hypothesized bit stream and the received bit stream for that state is used as the *path metric* for that state. The traceback unit records the most likely n state sequence leading to each state, encoded as one bit per state transition, logically organized as an n entry shift register where each entry is 2^k bits.

The path metric unit updates all 2^k entries for every 2 observations received from the demapper module. Once all the new path metrics are computed, the old path metrics can be discarded. As it computes each new path metric, it records the previous state pointer in the traceback unit.

After the path metric unit updates the values for all the states, the traceback unit follows the recorded previous state pointers and emits the bit corresponding to the oldest transition in the sequence.

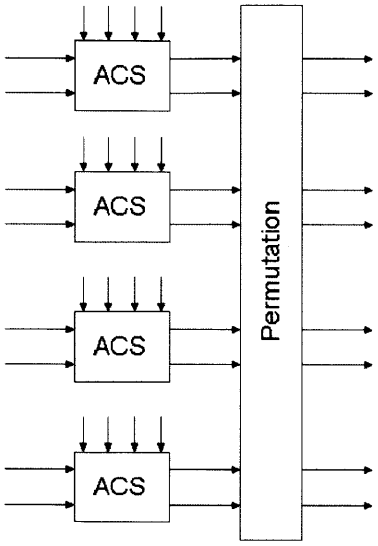


Figure 2-13: Path metric unit for k = 3

Path Metric Unit

The path metric unit, which is shown in Figure 2-13, contains one or more Add-Compare-Select (ACS) units for calculating the path metrics for each state. The ACS computes two path metrics at a time, as shown in Figure 2-14. The number of ACS units, which is a parameter to the path metric unit, controls number of path metrics updated per cycle.

The overall structure of the path metric unit is similar to that of a single FFT stage [22], with the FFT butterflies replaced by the ACS units. With the generalized pipelining technique presented in [22], we can easily parameterize the design of the path metric unit with the number of ACS units. This parameterization represents a tradeoff between area and power: the area increases as we increase the number of ACS units, while the power decreases.

Traceback Unit

The traceback unit contains a $n \times 2^k$ bit shift register and a decoder which reconstructs one bit at a time by traversing the most likely state transition sequence. Traversing $n = 35$ transitions in one cycle leads to long cycle times. To reduce the critical path,

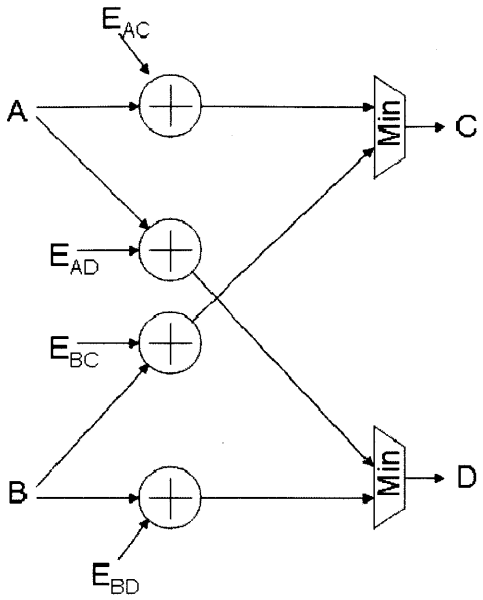


Figure 2-14: Add-Compare-Select unit

we pipelined the traceback unit, with a parameterized pipeline depth.

In the pipelined implementation of the traceback unit, a single pipeline stage in the decoder traverses s pointers. There are t such stages, such that $s \times t = n$ and $n = 35$. Each pipeline register needs to store one traceback memory column of 2^k bits and a current state index.

We varied the number of pipeline stages (1, 5, 7 and 35) to analyze various design alternatives. Figure 2-15 shows the area and power measurements for different pipeline depths. The results reflect the minimum frequency required to support the 54 Mbps bit rate for 802.11a. The figure also shows the number of bits written to the traceback memory per cycle and the complexity of the address decode logic for reading out the bits from the traceback memory, which is the input size of the multiplexing logic to read the data from each element of the shift register. The results show that the 5-stage decode consumes the least area and least power. The 1-stage version fails to meet the timing requirements.

Pipeline depth	Bit writes per cycle	Read complexity	Area (mm^2)	Frequency (MHz)	Power (mW)
35	35x64	2:1 mux	0.746	108	67.945
7	7x64	6:1 mux	0.713	108	52.310
5	5x64	8:1 mux	0.652	108	48.434
1	1x64	36:1 mux	Does not meet timing		

Figure 2-15: Viterbi synthesis results using TSMC 180nm library

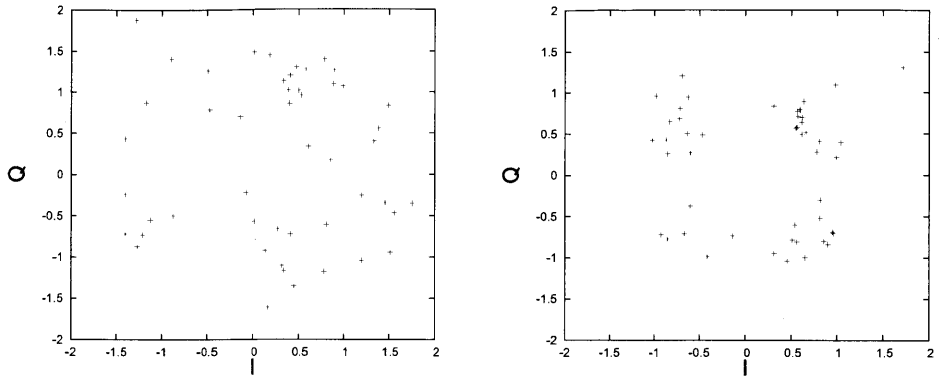
2.5 Parameterized Implementations of Other Computation Intensive Modules

Some modules, such as the LFSR Scrambler, are quite simple. Other modules, mostly in the receiver, employ complicated algorithms to help ensure quality reception. In the following subsections, we discuss the implementations of the more complicated and performance critical modules in the baseband processor.

2.5.1 Synchronizer

Our synchronizer implementation is based on the Schmidl-Cox algorithm [77], which synchronizes the receiver to the timing and frequency of the incoming signal. Schmidl-Cox detects the repeated preamble pattern by autocorrelation. If the autocorrelation is relatively high for an extended period, the synchronizer will decide the preamble structure has been observed. To make this determination, we employ a plateau detector which observes the autocorrelation value. Two criteria need be met for our plateau detector to assert the start of a packet: 1) the power of the autocorrelation divided by the square of the instantaneous power must be larger than an empirically determined threshold for many consecutive cycles and 2) the instantaneous power of the signal must be relatively stable throughout this period. The second condition exists primarily to compensate for interference.

Similar to other blocks in the transceiver, our synchronizer is highly parameterized. The configurable parameters include: 1) the length of the autocorrelation so that it can detect different preamble lengths, 2) the value of the threshold power ratio, 3) the number of cycles the threshold needs to be exceeded before declaring a detection, 4) the acceptable range for the power variation.



(a) Before Channel Estimation (b) After Channel Estimation

Figure 2-16: Subcarrier constellations for QPSK modulation before and after channel estimation. As we can see, our Channel Estimator is able to correct unrecognizable QPSK samples close to their desired positions

2.5.2 Channel Estimator

Fading in both frequency and magnitude can be quite extreme in OFDM systems. Figure 2-16 shows an example constellation from our transceiver before and after channel estimation. Although the fading makes the initial signal unrecognizable, our channel estimator recovers the familiar QPSK constellation.

We implement a channel estimator that can correct both amplitude and phase offset. The channel estimator first calculates the fading of the pilot sub-carriers. We determine phase offset with a CORDIC arctangent function. Detecting magnitude offset is more difficult because it requires both division and square root operations. We merge these operations and calculate the inverse square root of the pilot magnitude using the well-known Newton-Raphson method, which requires only scalar multipliers. Once the pilot fading is known, we apply a linear interpolation to determine the correction factors for the data carriers.

Our channel estimator is parameterized to: 1) the pilots' and guards' positions, 2) the pipeline length of the CORDIC and the inverse-square-root calculator.

Module	Lines of code		Area (mm^2)		Frequency (MHz)		Power (mW)	
	802.11a	802.16	802.11a	802.16	802.11a	802.16	802.11a	802.16
TX Controller	267	195	0.029	0.013	5	5	0.104	0.071
Scrambler	61		0.008	0.008	5	5	0.042	0.049
RS Encoder	–	105	–	0.027	–	5	–	0.114
Conv. Encoder	42		0.008	0.007	5	5	0.034	0.028
Puncturer	144		0.053	0.040	5	5	0.103	0.092
Interleaver	161		0.073	0.563	5	5	0.307	1.139
Mapper	89		0.420	1.719	5	5	1.572	6.346
Pilot	64		0.443	1.768	5	5	1.928	7.667
IFFT	318		4.736	11.651	5	5	11.961	38.319
CP Inserter	134		0.194	0.747	5	5	0.787	3.054
Synchronizer	1027		1.048	1.577	20	8	15.297	11.389
S/P	98		0.563	2.248	20	8	9.856	15.532
FFT	shared (IFFT)		4.526	10.733	5	5	11.431	35.300
Channel Est.	133		0.371	1.480	5	5	1.615	6.423
Demapper	202		0.303	0.828	5	5	0.627	2.328
Deinterleaver	shared (Interleaver)		0.212	0.779	5	5	0.900	3.375
Depuncturer	153		0.174	0.148	5	5	0.261	0.265
Viterbi Decoder	863		0.818	0.797	60	30	43.010	21.062
RS Decoder	–	45	–	0.007	–	5	–	0.027
Descrambler	shared (Scrambler)		0.008	0.008	5	5	0.042	0.049
RX Controller	153	86	0.321	1.263	5	5	2.424	9.638
Libraries	2163		–	–	–	–	–	–
Parameters	472	565	–	–	–	–	–	–
Total	6544	6648	14.308	36.411	–	–	102.301	162.267

Figure 2-17: Synthesis results for modules in 802.11a and 802.16 transceivers using the TSMC 180nm library. Area and power estimations are generated by Synopsys Design Compiler. The areas for the 802.11a and 802.16 designs are equivalent to 1.4M and 3.6M two-input NAND gates respectively.

2.6 Results

We wrote the designs of all the transceiver components in Bluespec. We verified each component using testbenches written in Bluespec. Simulations were carried out using Bluesim, which is a cycle-accurate simulator for Bluespec designs.

We generated a variety of RTL transceiver components for both the 802.11a and 802.16. All the components were generated from our OFDM library by passing appropriate parameter values.

The RTL was generated using the Bluespec Compiler (version 3.8.69), and then synthesized with Synopsys Design Compiler (version Y-2006.06) with the TSMC 180nm standard cell libraries. Figure 2-17 shows the post-synthesis area and power estimates as well as the clock frequency for the design to meet its respective standard. The power estimates were statically generated by Design Compiler.

The lines of code for each parameterized module and general libraries that we

implemented are given in the table. The entire code has less than 8000 lines.

The 802.11a transceiver code takes 6544 lines of code, while the 802.16 transceiver code takes 6648. Of this, more than 85% of the code is shared. This gives us evidence of how much code we can leverage from the common library when implementing a different protocol.

2.7 Related Work

Dave *et al.* [22] discuss the microarchitectural exploration of an 802.11a transmitter via synthesizable highly-parameterized descriptions in Bluespec. They explore various microarchitectures of the IFFT, which is the most resource-intensive module of the 802.11a transmitter. Our paper is in the same genre but shows a much more elaborate use of parameters, primarily motivated by IP reuse.

Nordin *et al.* [64] present a parameterized generator for Discrete Fourier Transform (DFT). The generator accepts parameters like the input size of the Fourier transform as well as microarchitectural parameters that control the concurrency in the generated DFT cores. This approach is similar to our approach in the sense that it enables parameterization of both algorithmic and microarchitectural exploration. The approaches differ in that they rely on scripts and other techniques to generate Verilog programs, whereas we rely on the parameterization capability of the hardware-description language itself. The other difference is the scope of the parameterization — we have parameterized the whole OFDM transceiver, whereas [64] does so only for DFT. Similar remarks apply to Zhang *et al.* [96], who present a framework to enable algorithmic and architectural co-design for interference suppression in wireless receivers.

Salefski *et al.* [75] show how reconfigurable processing can meet the needs for wireless base station design while providing the programmability to allow field upgrades as standards evolve. This is an orthogonal concern to the sharing of code across multiple protocols.

Brier *et al.* [12] show how C/C++ models can be used for architectural exploration

and verification of DSP modules. It proposes guidelines for building C/C++ models that aid in the verification process.

Hourani *et al.* describe domain specific tools for the signal processing [41]. These tools automatically generate different architectural variations for signal processing algorithms, enabling algorithm experts who are not skilled hardware designers to make area/performance/power tradeoffs.

Krstic *et al.* [53] present a VHDL implementation of low power 802.11a baseband processor. They show the post-layout area and power estimations of their implementation which is synthesized with their 250nm standard cell library. Our high-level parameterized Bluespec implementation is comparable to their dedicated VHDL implementation in terms of area and power.

2.8 Summary

Power and cost constraints dictate a need for specialized circuits in the burgeoning market of handheld devices and sensors. Yet, the ever increasing chip design costs and time-to-market of ASICs creates a major hurdle in the exploitation of this opportunity. We think that parameterized reusable components are the most immediate solution to this problem. In this chapter, we have shown that various components for OFDM-based wireless protocols can be created in a manner so that they can be instantiated with appropriate parameter values to be part of different protocols. A powerful library of OFDM-based components can dramatically reduce the cost of implementing OFDM-based protocols and can also facilitate algorithmic exploration of new protocols. We already have a set of components that are rich enough to implement both 802.11a and 802.16 transceivers. Futhermore, Pitakenen *et al.* [71] were able to extend it to support 802.15.3 (WUSB). As a result, we are confident that our library is rich enough to implement many other OFDM protocols.

This type of component library development depends on a language like Bluespec SystemVerilog, which has the necessary type system and static elaboration facilities to make this level of parameterization feasible. One also needs a language with proper

modular composition. Furthermore, without the ability to synthesize the designs, meaningful evaluations of these designs would be impossible.

In the next chapter, we will discuss how we can leverage our library to implement a FPGA-based wireless platform for wireless experimentation.

Chapter 3

Airblue: An FPGA-based Platform for Cross-Layer Protocol Development

In this chapter, we present Airblue, an FPGA-based wireless platform designed especially for cross-layer experimentation. It supports speeds comparable to commodity 802.11 hardware but is designed with modular refinement in mind. We discuss two properties, namely *latency-insensitivity* and *data-driven control*, that are essential for modular refinement. To facilitate cross-layer protocols implementations which have tight latency requirements, Airblue is designed to pass additional information up from the PHY layer to the MAC layer using *annotated streams* and to let the MAC reconfigure the PHY in sub-microseconds. The streaming interfaces between the two layers allow data to be processed as soon as it is ready, thus amortizing the processing latency by overlapping it with the communication latency. Finally, we show how Airblue can be extended easily to become a high-speed FPGA-accelerated simulator with software channels. The key to this extension is our ability to virtualize different features from different FPGA platforms to provide the same set of abstract interfaces.

The content of the next two chapters is based on the publications presented at the 2010 ACM/IEEE Symposium on Architecture for Networking and Communication Systems (ANCS) [62] and the 2011 IEEE International Symposium on Performance

Analysis of Systems and Software (ISPASS) [26]. Kermin Elliott Fleming and I are the main contributors in the Airblue project. Kermin focused on FPGA optimizations and bring-up activities and I focused on the design principles and the baseband algorithmic optimizations. *This research was funded by NSF under grants CNS-0721702, CCF-0541164, and CCF-0811696, by a Cisco URP, and by Nokia.*

3.1 Introduction

In recent years, researchers have developed a large and growing set of protocols and algorithms to improve the throughput and capacity of wireless networks. These schemes span the physical (PHY), MAC, and network layers of the protocol stack. Some examples include interference cancellation [37], ZigZag decoding [30], Conflict Maps (CMAP) [92], the SoftPHY interface [46, 45, 91], SampleWidth [61], Analog Network Coding [49], MIXIT [50], VWID [33], ODS [32], and SWIFT [73].

A common theme in all these schemes is that they embody some form of *cross-layer design*, i.e., additional information is passed from lower to higher layers and higher layers exercise some control over lower-layer decisions. For example, the SoftPHY interface [45] extends the receiver PHY to send to higher layers confidence information about each bit's decoding, so that those layers can perform better error recovery [46], bit rate adaptation [91], diversity routing [50], and so on. In fact, even the simple example of using the receiver's signal-to-noise ratio (SNR) to determine a transmit bit rate is an example of PHY-MAC cross-layer information. Given the strong real-world interest in high-speed wireless networks, we expect a significant amount of continuing research in the area of cross-layer protocols.

The effort required to implement any high-performance wireless protocol from scratch is enormous. Therefore, one would like to start from a base implementation of, say, 802.11. The problem with this approach is that commodity hardware implementations offer no opportunity for making changes while the Software Defined Radios, like GNUradio [29], do not offer sufficient performance for cross-layer experiments. Platforms like WARP [93] and SORA [86] can provide high speeds but are

still quite difficult to modify for cross-layer experiments, as we show later.

This motivated us to implement Airblue, a high-speed platform that is also easy to modify for cross-layer experiments. As we already had a set of previously implemented ASIC OFDM modules described in Chapter 2, we decided to implement our designs on an FPGA platform with a radio frontend. Unfortunately, many of the ASIC modules did not meet the required performance on the FPGA. For example, our ASIC Viterbi decoder only achieves 1/3 of the required throughput when synthesized for FPGA, forcing us to redesign the decoder.

Despite having to modify most of the major modules, we were able to complete a functional design in just 5 months. We attribute our rapid development time to the two properties that are essential for *modular refinement*, i.e., the ability to make changes in one module of a system without having to understand or make changes to the rest of the modules in the system. These properties are *latency-insensitivity* and *data-driven control*. Our successful experience leads us to believe that these properties are also useful for cross-layer protocol implementation. Moreover, we find that the need for these properties also shows up in pure software implementations on multicores.

In Airblue, we also implement a hardware MAC in additional to the PHY. They communicate between each other through streaming interfaces operating at byte granularity. This contrasts with the conventional approaches which operate at packet granularity. The advantage of the streaming interface is that the destination node can start processing cross-layer information as soon as they are available.

We believe that Airblue on FPGAs represents an ideal platform for the development of new wireless protocols. First, it is much easier to describe the fine-grain parallelisms present in DSP algorithms in a high-level hardware description language like Bluespec than in sequential software language like MATLAB or C/C++ (a chapter is dedicated to discuss this in detail in Chapter 6). Second, most protocols are going to be implemented in ASIC hardware to meet the power, throughput, and latency requirements necessary for deployment. In order to do so, the hardware implementations of many algorithms in a wireless system can only be approximations of the

originals. Common approximation techniques that might be applied include: 1) using fixed point arithmetic instead of floating point arithmetic; 2) replacing complicated arithmetic with a simplified one; 3) replacing full-block processing with a sliding window approach; 4) ignoring less significant terms in the algorithms. In general, these approximations distort the input and the behavior of downstream modules in ways that are difficult to quantify. Therefore, by forcing the developers to implement the protocols in hardware, Airblue allows them to study the impact of approximations on the performance on the proposed protocols in the early phase of the development process.

Chapter organization: We first discuss several concrete examples of cross-layer protocols (Section 3.2), and identify three requirements for each protocol: the information that needs to be conveyed from the lower to higher layers; the dynamic reconfiguration of the lower layer required by higher layers; and the modules in the PHY or link layers that need to be changed substantially. In Section 3.3, we discuss the properties an implementation must have for modular refinement, regardless of whether it is implemented in hardware or software. In Section 3.4, we describe the Airblue platform and our implementation of the 802.11 physical and link layers. In Section 3.5, we describe how we extend Airblue to become a high-speed fpga-accelerated simulator by replacing the Radio Frequency (RF) device with a software channel. We discuss the limitations of Airblue in Section 3.6, and summarize in Section 3.7.

3.2 Cross-Layer Protocols

The research literature has many examples of cross-layer wireless protocols, which are characterized by the use of information from higher or lower layers to achieve performance gains. Conceptually, most of these ideas can be implemented by extending existing standards like 802.11. We survey some examples of such protocols in this section, and identify the cross-layer interaction requirements as well as the modifications needed on top of an 802.11 implementation. The examples are summarized in

Protocols	Lower-to-Higher Layer Information	Higher-to-Lower Layer Configurations	PHY/MAC Modifications
ZigZag	channel characteristics	sender MAC identity	New PHY decoder
PPR	per-bit confidences	sub-packet retransmission	Replacing the hard-decision decoder with a soft-decision decoder
UEP		symbol-level encode/decode	
RBAR	per-packet SNR estimates	link-layer feedback to sender	Calculate per-packet SNR
SoftRate	per-bit confidences	link-layer feedback to sender	Replacing the hard-decision decoder with a soft-decision decoder
CMAP		link-layer feedback to sender, quick switch rx-to-tx	Early MAC header decode
FARA	per-carrier SNR estimates	link-layer feedback to sender, per-carrier modulation	Calculate per-carrier SNR

Figure 3-1: Examples of cross-layer protocols and their implementation requirements.

Figure 3-1.

Interference cancellation: Interference cancellation is a popular technique at the physical layer to decode multiple transmissions simultaneously. *ZigZag* [30] combats interference by using two instances of two collided packets, the second instance comes from a retransmission, to recover each of the individual packets. Implementing ZigZag requires new decoding logic in the 802.11 PHY, and some exchange of information between MAC and PHY. For example, the ZigZag decoder in the PHY must know the MAC address of the sender *while* it is decoding a packet, in order to track the sender-specific frequency offset and compensate for it.[1]

Error recovery: Improving error recovery algorithms and modifying them to suit application requirements can greatly increase application throughput. We consider two examples. Most link layers in wireless data networks retransmit entire frames on even a single bit error. In contrast, the PHY in *partial packet recovery (PPR)* [46] computes and exports per-bit confidence information or *SoftPHY hints*, using which the link layer identifies and requests a retransmission of only those bits with low confidence. *Unequal error protection (UEP)* [47] is another example of a technique that modifies the standard error recovery algorithms. It is known that an application's throughput and loss rate requirements can be better met by allowing the application to control the mapping from the data payload to the PHY modulation and coding.

[1]This way of implementing ZigZag is different from the description in [30], which does not attempt to preserve layering.

For example, video applications may be able to tolerate some bit errors as long as the high-priority bits in the video stream are encoded robustly. With UEP, the application specifies the priority of bits in a payload to the lower layers, and triggers reconfiguration of the PHY modulation and coding schemes multiple times within a packet.

Bit rate adaptation: Wireless physical layers today can transmit data at multiple different bit rates by varying the modulation and coding of the transmission. Many bit rate adaptation protocols use PHY information to quickly estimate the channel quality and pick a "good" bit rate. For example, the MAC in *RBAR* [40] uses per-packet SNR estimates from the receiver PHY to pick the bit rate for the next packet. Alternatively, *SoftRate* [91] picks the bit rate using estimated bit error rate (BER) computed by per-bit SoftPHY Hints. AccuRate [79] uses per-symbol dispersions computed in the PHY demodulator to estimate channel quality and pick transmit bit rates. In all these protocols, the PHY at the receiver passes up extra information (e.g. SNR estimates, SoftPHY hints, symbol dispersions) to the MAC, and an appropriate feedback is sent to the MAC at the sender in a link-layer feedback frame. The transmitter's MAC then reconfigures the PHY to transmit at the suitable bit rate.

Concurrent transmissions: The popular Carrier Sense Multiple Access (CSMA) MAC protocol avoids transmission when the senders sense a busy channel. In contrast, the *CMAP* MAC protocol [92] uses additional information about *who* is transmitting on the channel. With this information, a CMAP node can send its packet if its transmission will not significantly interfere with the ongoing one. CMAP can be implemented efficiently using two cross-layer primitives. First, the PHY streams the MAC-layer header as soon as it is received, enabling the MAC to identify the sender and receiver of an ongoing transmission before the transmission completes. Second, if the MAC believes that its transmission does not interfere with the ongoing transmission, the MAC instructs the PHY to quickly stop receiving the ongoing transmission and switch to transmit mode.

Variable width channel allocation: Some MAC protocols allocate channel resources not only in the time dimension but also in the frequency dimension. For

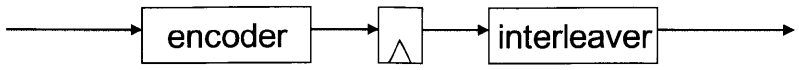
example, *FARA* [72] allocates frequencies to each user based on the SNR estimates over various sub-bands to a particular user, because different users might see different fading effects over the transmission frequency band. Other research [15, 33] allocates disjoint frequency bands to different senders to mitigate interference, with the width of the channel depending on the received signal strength from the sender. In all these protocols, the PHY needs to communicate per-subchannel signal quality information up to the MAC layer for each packet. The MAC must be able to instruct the PHY to send and receive data over a particular subset of the available frequencies.

3.3 Implementation Challenges

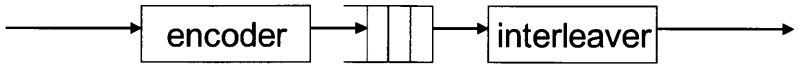
The previous section outlined the enhancements required to a base implementation to implement various cross-layer protocols. The degree of difficulty in making these changes depends largely on what the base system provides and how it is implemented. At one extreme, if the base system is implemented with a rich set of interfaces and parameters, the implementation of a new protocol may just be a matter of setting some configuration parameters. At the other extreme, the system may be implemented in such a way that changing one module may require a deep understanding of the whole implementation, triggering adjustments to many other modules. If the effort required to implement a new protocol is substantial, then the platform is not appropriate for cross-layer protocol experimentation. In this section, we discuss the design decisions of high-performance physical layer designs that directly affect our ability to modify them.

3.3.1 Latency-insensitive Designs

Consider an encoder in a hardware PHY that applies an error-correcting code to a stream of bits, and feeds the resulting bits into an interleaver that shuffles the bits, as shown in Figure 3-2(a). Suppose the encoder writes a symbol once every clock cycle into a register, which the interleaver reads in the next cycle. Now suppose a designer modifies the encoder to use a more complicated code that requires two clock



(a) LS - The interleaver connected to an encoder assumes a valid input is written into the register every clock cycle.



(b) LI - The downstream interleaver waits for input to arrive via a FIFO from the upstream encoder.

Figure 3-2: An example to contrast latency-sensitive (LS) and latency-insensitive designs (LI).

cycles to encode the bits instead of one. This modification to the encoder compromises the correctness of the interleaver, which must now be modified to account for the fact that its input comes in only every other cycle. The problem gets harder if the error-correction code takes a variable amount of time. Such designs where modules implicitly make assumptions about the latencies of the other modules are called *latency-sensitive* designs. The biggest problem in modifying latency-sensitive designs, for example, in Rice WARP [93], is that it is difficult to know the assumptions of the original designer by examining the design. Furthermore, experience strongly suggests that it is practically impossible to document all such assumptions in a real-world design.

Modules in *latency-insensitive* designs, on the other hand, do not make assumptions about the latencies of the other modules in the pipeline — a data transfer occurs only when the upstream module has produced enough data and the downstream module is ready to consume it. To execute modules in parallel, finite-sized FIFO queues are added between modules. Figure 3-2(b) shows how the designs in Figure 3-2(a) can be made latency-insensitive by adding FIFOs. Latency-insensitive designs are in general easier to modify than latency-sensitive designs. However, converting a latency-sensitive design into a latency-insensitive design is quite difficult after the fact because the designer's latency assumptions are not known. Latency-insensitive

designs have a further benefit that they make it easy to independently tune various modules for higher performance, without exacerbating the verification problem of the complete design.

Latency-insensitivity in software implementations: Latency-sensitivity issues show up quite differently in software because software is almost never written to describe clock-cycle by clock-cycle behavior. Programmers write software to process events assuming that the underlying machinery (e.g., processors, caches, I/O) is fast enough to do the required work in time. If the underlying machinery is not fast enough, then the implementer has two choices: buying faster machinery or optimizing the programs.

Optimizations in performance-critical systems are done generally in two ways. First, there are algorithmic optimizations to take advantage of machine specific microarchitecture. For example, data structures might be modified to fit into a particular cache line size. Second, one can apply static thread scheduling and static resource allocation techniques to achieve efficient multiplexing of the underlying machine resources. The allocation issue is further complicated in current multicore systems: processors may be multiplexed by allocating separate cores to separate threads, but the programmer has essentially no control over shared resources like caches and on-chip networks. This lack of control introduces the possibility of unpredictable interactions between different code components, and often causes high variability in performance. Highly tuned systems are very brittle with respect to performance — small changes in a single module can have a deep effect on the performance of the whole system. For example, increasing the size of a data structure or changing a code path in a single module can ripple through the system causing a cascade of unexpected cache misses in other performance critical modules. Sometimes unforeseen performance changes can be caused by just switching to a newer compiler version. In systems with tight timing requirements, like WiFi, the delays may be unacceptable, forcing the programmer down the painful path of modifying large portions of the system to regain lost performance. In short, such systems, even though they are written in software, are often unmodifiable in practice.

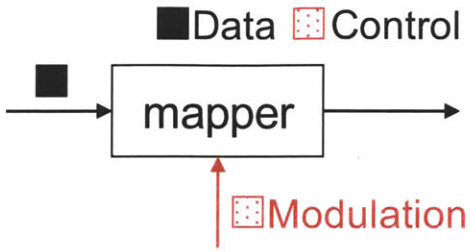
Modifications of pipelines with static scheduling and static resource allocation also cannot be undertaken without a deep understanding of the system. Because it is practically impossible to document every assumption that goes into a high-performance implementation, parallel software wireless platforms like SORA are hard to modify.

Until now, we have considered only the problem of executing a set of independent tasks on a multiplexed substrate. Achieving high performance in the context of *communicating* processes with shared data is even more difficult. Consider the relatively simple sub-component of an OFDM pipeline shown in Figure 3-2. If both the Encoder and the Interleaver execute sequentially, i.e., in one thread, then the correctness would not be affected by the changes in the code of either module. Scheduling the modules in the pipeline, e.g., when we switch from Encoder to Interleaver, is done statically and is part of the user code. Furthermore, information is usually passed from one module to another via shared memory using pointers. But if, for performance reasons, we want to execute the Encoder and the Interleaver in parallel, then accesses to the shared data structures (FIFO queues) have to be coordinated using locks. Since locks are expensive, software solutions minimize locking by doing coarse-grained synchronization.

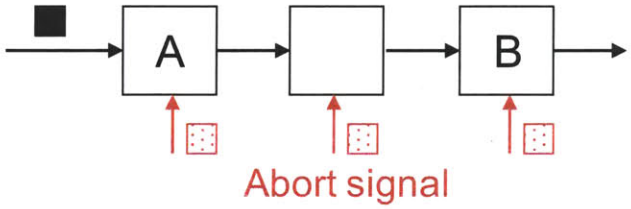
For robust and modular software implementations, data should be passed from one module to another as messages via message-passing ports, rather than shared memory. This way, a module can modify the (local) data without having to lock it. The message-passing protocol needs to guarantee that the recipient has enough buffer before the producer sends data. The scheduling has to be dynamic enough to deal with variable processing times. Unfortunately, in current multi-core architectures, dynamic scheduling of modules requires prohibitively high communication and synchronization between processors.

3.3.2 Synchronizing Data and Control

Cross-layer protocol stacks require new ways of reconfiguring the lower layers at run-time, unlike standard commercial implementations in which specific configurations



(a) Input data bits and corresponding modulation must arrive together at the mapper, which modulates data into PHY symbols.

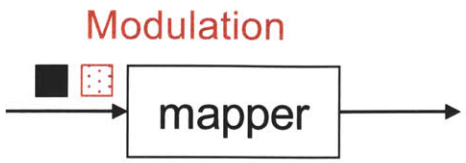


(b) When aborting the reception of a packet at the PHY, new data should not be sent into the pipeline until reconfiguration is complete at all the modules.

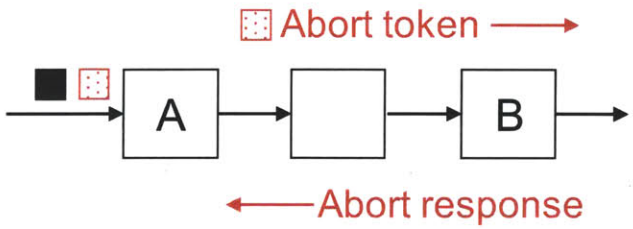
Figure 3-3: The problem of synchronization between control and data when reconfiguring a lower layer.

and control paths are embedded in the design. The commands from the higher layer to trigger reconfiguration are usually referred to as "control" to distinguish them from the actual data through the pipeline.

Consider the mapper module shown in Figure 3-3(a), which takes a data input (a group of bits to map into a PHY symbol) and a control input (the modulation that determines the mapping). For the mapper to function correctly, the modulation control should arrive at the same time as the data bits it applies to. Sometimes the reconfiguration affects several modules and has to affect them in a *coordinated* way. For example, the CMAP MAC protocol requires rapid switching from receive to transmit, where the ongoing reception must be aborted and all modules must prepare to transmit. For correct operation, transmit data should be sent along the pipeline only when all the modules have finished processing the control signal to abort and flush (Figure 3-3(b)).



(a) The modulation control and data bits arrive together along the same input at the mapper.



(b) Aborting a reception is accomplished by forwarding an abort token along the datapath and withholding the data after the token until reconfiguration is complete and a response is received.

Figure 3-4: Examples illustrating data-driven control.

Why is synchronization a hard problem when one wishes to add a new control? In typical hardware designs, the processing latencies of the different blocks are computed *a priori* and control is designed to be sent to each block just in time for the data; as a result, synchronization between the control and data is implicit. Although such designs can achieve high performance because one need not expend circuitry to handle synchronization, they are also hard to modify. Adding new control requires a careful understanding of the latencies of all the modules in the pipeline. Therefore, we use a different solution: *data-driven control.*

With data-driven control, messages between blocks contain *both* control information and the set of data values the control must operate on. Control *tokens* are embedded into the datapath along with the data, and are not modified by blocks that do not need to act on them as the message flows through the pipeline. The control information is stripped off when the message leaves the pipeline. Control tokens can be interspersed with data at any granularity, enabling us to pass control

with every bit or groups of bits, or per-packet. This approach incurs the overhead of extra hardware circuitry to pass and identify control tokens through the datapath, but allows protocol designers to modify the structure of pipelines or refine any individual block easily without worrying about retiming the controls. Figure 3-4 shows an implementation of the examples in Figure 3-3 with data-driven control.

The concept of data-driven control is neither new to hardware systems nor to software systems. For example, packet transmissions in wormhole networks use header-flits (controls) to reserve buffers of each node along the paths until all the following body-flits (data) pass through that node. Another example is that Click [52] uses "packet annotations" to couple control and data together. The notion is also used in SDR-based systems [65].

3.3.3 Passing Information to Higher Layers

Information from lower to higher layers may be passed at any granularity—once per bit, once per group of bits, or once per packet. As one may expect, passing new information along the pipeline faces an association problem similar to that between control and data discussed in Section 3.3.2. Meeting stringent latency requirements when passing up information is also a challenge because most network stacks assume that the higher layers act on data at the granularity of a frame at a time and do so only after the lower layer finishes its processing for the entire frame. This coarse granularity of processing can be attributed to the prohibitive cost of fine-grained communications in software.

To pass information between layers in a timely manner, we propose using a *streaming* interface between the layers. For example, when the MAC and PHY are both implemented in hardware, the PHY can send up bits to the MAC as they are decoded, instead of waiting for the complete frame to be received. This way, the MAC can receive and act on PHY information in a timely manner. Extra information along the streaming interface can be passed up using *annotations*. An annotation is additional information that is sent in-band along with the data. For example, when the PHY computes per-bit SoftPHY hints, the hints are pushed through the datapath along

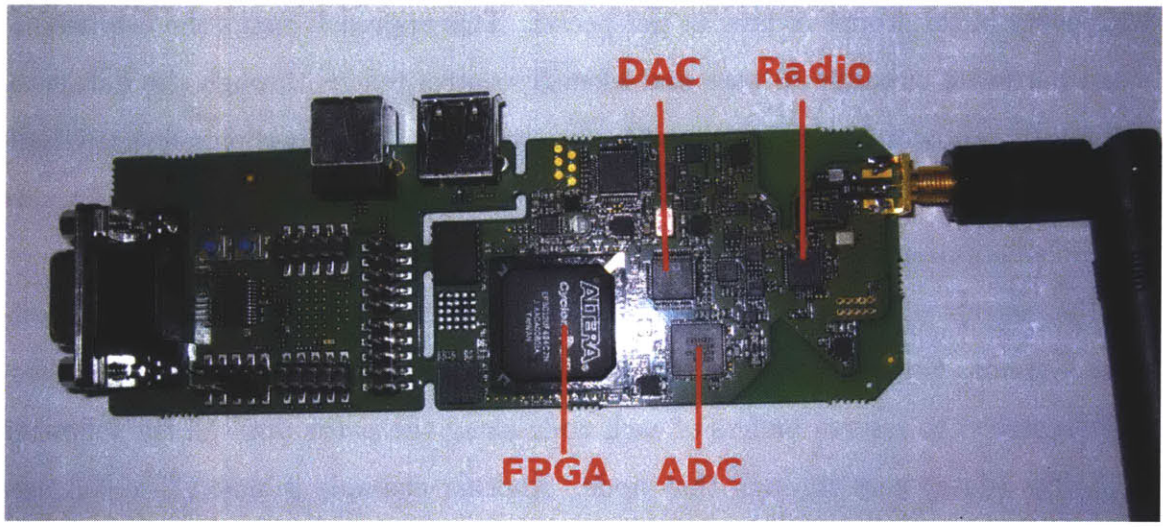


Figure 3-5: Airblue hardware.

with the corresponding bits.

3.4 The Airblue Platform

Our platform presently consists of a MAC and PHY implemented on an FPGA. In the future, we plan to integrate these layers with the higher layers of the networking stack by either exposing the FPGA as a network device on a Linux PC or by implementing the higher layers in software on the FPGA. In this section, we first describe the hardware and software components of Airblue and then present some performance results.

3.4.1 Airblue Hardware

Figure 3-5 shows a picture of the FPGA platform on which Airblue is implemented. The platform is developed by Nokia Inc. It consists of an low-end Altera Cyclone III FPGA. The FPGA has a direct connection to a 2.4 GHz RF front-end capable of 20 MHz and 40MHz baseband modulation, and communicates with the host processor using high-speed USB.

Figure 3-6 shows a block diagram of the system. The system is divided into three clock domains at 20 MHz, 25 MHz and 40 MHz respectively. The *Device Interface*,

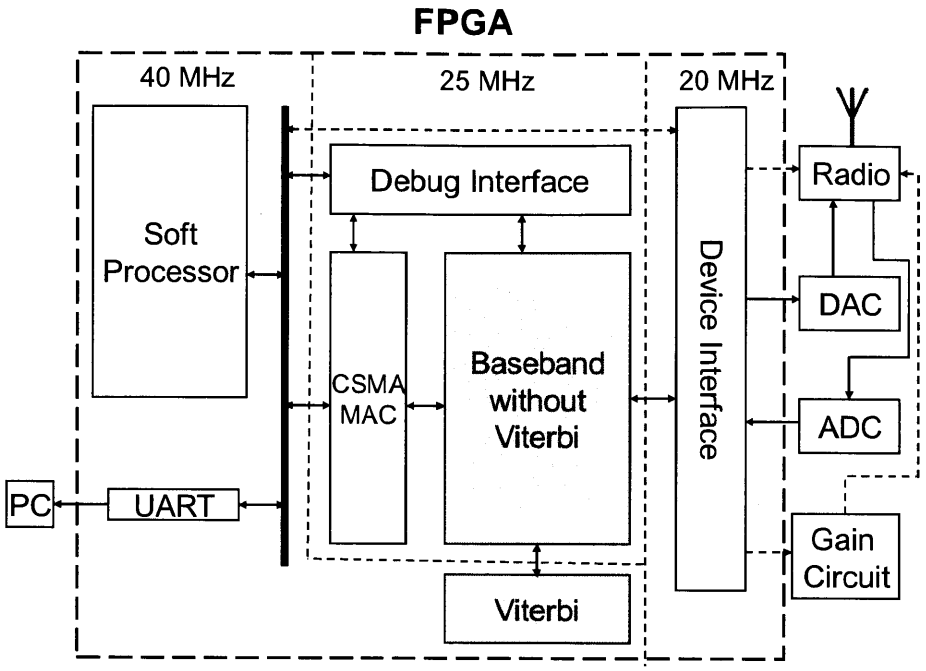


Figure 3-6: AirBlue system architecture.

clocked at 20 MHz, provides a generic interface between the digital baseband and the RF front-end.

There are three blocks—the *Baseband Processor*, the *MAC Unit* and the *Debug Interface*—in the 25MHz clock domain. The *Baseband Processor* implements the PHY, converting digital bitstreams to digital baseband signal during transmission and performing the inverse during reception. The *MAC Unit* controls when the *Baseband Processor* can transmit or receive and implements an acknowledgment protocol. The *Debug Interface* collects internal state of other blocks and communicates this state to the host PC.

The *Soft Processor*, running at 40 MHz, handles off-chip communications to a host PC through USB. In the future, we plan to use the *Soft Processor* to execute the software implementing the protocol layers above the MAC layer.

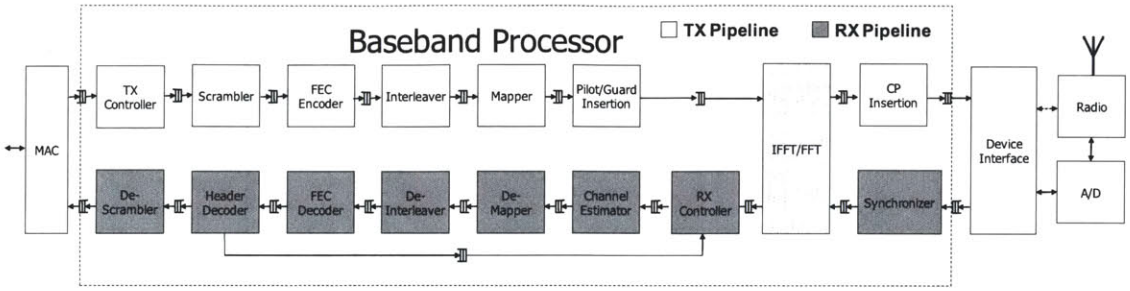


Figure 3-7: OFDM baseband data flow in Airblue.

3.4.2 Baseband Processing on FPGA

As mentioned before. our baseband design (shown in Figure 3-7) is based on the OFDM workbench presented in Chapter 2 that targets ASICs (not FPGAs). To make the system meet our performance targets while fitting on FPGAs, we increase the concurrency of some blocks (e.g. Viterbi, Synchronizer) so that we can run at lower clock speed, i.e., 10s MHz, as well as decrease the resource usage by time-sharing other blocks (e.g. FFT and IFFT). Although most of the modules are modified at some point, we are able to successfully obtain a functional system in 5 months thanks to the fact that all the original modules are latency-insensitive.

In developing new protocols, users may have to modify the baseband modules in Airblue for the following reasons.

New features: A module may have to be modified to provide additional features. Modifications can be as simple as exposing some internal state of a module to other modules, or can be substantial modifications needed to implement new algorithms like computation of SoftPHY hints (see Chapter 4).

Algorithmic modifications: Users may need to replace under-performing algorithms with more sophisticated ones. For example, the channel estimator in the original library turned out to be inadequate because it had never been tested with a real radio. It performed minimal phase tracking and no magnitude scaling, and had to be re-implemented using an algorithm that performed both.

Performance tuning: A module may have to be modified to meet tighter throughput or latency constraints. Modifications normally involve exploiting more parallelism

in an algorithm or improving data movement between modules. For example, we increased the per-cycle throughput of the original FEC Decoder twice to compensate for performance loss due to the lower clock frequency in FPGAs as compared to ASICs.

FPGA-specific optimizations: Some hardware structures targeted for ASIC implementations do not map well onto the FPGAs and have to be modified to meet resource usage and timing requirements. A typical example involves mapping register banks onto more dense SRAM-like resources. Conversely, the FPGA contains primitive resources, like multipliers, that enable the development of more robust and efficient algorithms.

3.4.3 Hardware Streaming MAC

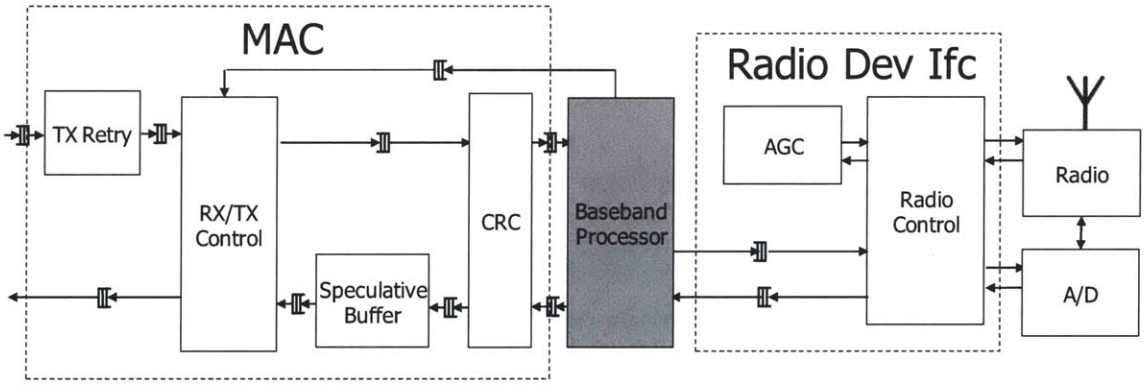


Figure 3-8: AirBlue's MAC and Radio Device Interface. Our MAC consists of (i) *RX/TX Control*, which handles the 802.11 transmission control protocol, including packet acknowledgments and inter-frame timings; (ii) *TX Retry*, which buffers the transmitted packet until it is acknowledged; (iii) *CRC*, which handles CRC checksums of outgoing and incoming packets; and (iv) *Speculative Buffer*, which stores incoming packets until their CRC checks pass. Radio Device Interface consists of (i) *Automatic Gain Control (AGC)*, which ensures received signals cover the full dynamic range of the ADC; and (ii) *Radio Control*, which configures the Digital-to-Analog Converter (DAC), Analog-to-Digital Converter(ADC), and RF circuits.

The MAC is responsible for determining when the baseband sends or receives data. A typical MAC transmits frames between higher layers and the baseband, and reacts on receiving a frame by, say, sending an ACK back to the source.

Airblue's MAC has two important properties that enable it to support a larger range of protocols than traditional MACs. First, the MAC is implemented in hardware with dedicated low-latency channels to the baseband. This approach allows the MAC and the baseband to communicate large amount of data back and forth with tight latency. Second, it communicates with the baseband in a streaming manner, i.e., at the granularity of bytes instead of frames, enabling the MAC to start processing data from the baseband as soon as it is decoded. These two properties are necessary for the implementation of cross-layer protocols that require the MAC and the baseband to frequently communicate with each other in a timely manner.

The architectures of the MACs can differ vastly depending on their access policies. For example, a MAC implementing CSMA will look completely different from a MAC implementing time division multiple access (TDMA). Airblue provides an 802.11-compliant CSMA MAC, as shown in Figure 3-8, that is modular enough to facilitate the implementation of derivative MAC protocols.

3.4.4 Radio Device Interface

To send and receive on-air signals, the baseband must communicate with external devices like DACs, ADCs, gain circuits (see Figure 3-8). A challenging implementation problem is that many of these components are latency-sensitive. For example, if we change the gain, it takes a certain number of cycles before the correct gain is reflected in the incoming samples. To make matters worse, components implementing the same functions from different vendors have different timing characteristics. To keep the baseband flexible, we abstract the physical platform as a pair of bidirectional FIFOs to which the baseband can connect. From the baseband's perspective, the incoming FIFO provides radio samples from the physical radio receiver and the outgoing FIFO sends samples to the physical radio transmitter.

3.4.5 Development Environment

Airblue has been developed using Intel's architect's workbench (AWB), an open source design management tool [3]. AWB provides a graphical interface for configuring, building, and running FPGA/software co-designs. The following properties of AWB allows Airblue users to rapidly assemble various wireless models for evaluation.

Automatic Multi-Clock Support: In the baseband implementation, the throughput of each module in the pipeline may not necessarily match if the whole design is running at the same clock frequency. As a result, the peak performance of the whole pipeline can be bottlenecked by a single slow module. In DSP systems, this rate matching issue, if not addressed, can greatly reduce performance. AWB solves this problem by providing automated support for multiple clock domains. A user can change the throughput of a module by specifying a desired clock frequency. AWB will automatically instantiate an FPGA primitive providing the specified clock frequency and add special cross-domain communication constructs between every pair of connected modules that are in different clock domains. We achieve this service by extending the mechanisms used by the SoftConnections [69] design tool to carry clock information. In practice, multi-clock support improves modularity. In a typical hardware design users must either pollute their module interfaces to supply submodules with clocks, or the submodule must know its parent's clock frequency to synthesize its own clock. Neither of these cases is portable. Because our compilation tools handle multiple clock domains, Airblue's modules gain a degree of portability.

FPGA Virtualization: In principle, Airblue can be executed on an FPGA platform as long as the hardware design fits into the FPGA and there are two pair of bi-directional links: one between the FPGA and a RF frontend and another between the FPGA and a host processor. In reality, various FPGA platforms have different interfaces to the RF frontend and are connected to the PC through different types of links, e.g., PCI-E and USB. Each type of link requires specific RTL and possibly software codes to be run on both sied of the link. Users should be insulated from these details. We implement Airblue on top of LEAP [68], which is a collection of device

drivers for specific FPGA platforms. LEAP provides a set of uniform interfaces across devices like memory and off-FPGA I/O. It also provides automatic mechanisms for multiplexing access to these devices across multiple user modules. For example, as mentioned before, the radio device interface is abstracted as a pair of bidirectional FIFOs which are used by the baseband processor to receive or send baseband samples.

By porting Airblue modules to LEAP, we gain portability and modularity. Airblue models can be run automatically and without code modification on any platform supported by LEAP and providing LEAP I/O functionality, including future high-speed radio platforms. Because LEAP handles multiplexing of these resources automatically, user modules are insulated from one another in sharing common devices, aiding in modular composition.

Plug-n-Play: In general, Airblue provides multiple implementations of each module. In many cases, users want to experiment with different combinations through mix-and-matching different implementations. Users may also wish to use their own modules in combination with existing ones. While this can be achieved by modifying the source code, this sort of work is usually tedious and, therefore, prone to error. To facilitate this process, AWB provides GUI support for plug-n-play designs. AWB users pick the implementation of each module by choosing from a list of available implementations. This plug-n-play approach greatly increases the speed of constructing a working wireless system. Moreover, the plug-n-play property facilitates the debug process because any system-level testbench can be used to test new modules by plugging it directly into the rest of the system. This methodology has been shown to be effective in producing new designs rapidly [21].

3.4.6 Baseline Performance

We have implemented an 802.11g transceiver capable of sending data at the 6, 9, 12, 18, and 24 Mbps bit rates. We have also implemented various cross-layer mechanisms on Airblue, as described in Section 4.2.4.

Airblue's throughput and latency: To understand Airblue's performance better, we evaluated the baseline 802.11 implementation using a pair of nodes, one configured

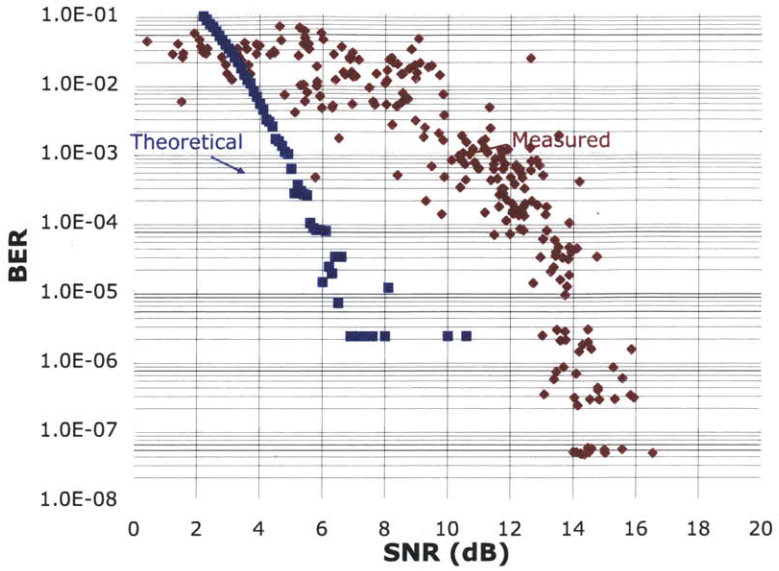


Figure 3-9: BER vs. SNR for 12 Mbps.

as a transmitter and the other as a receiver. We used two topologies. To assess the SNR vs. BER (Bit-Error Rate) performance, we attached one node to a cart and wheeled it around to vary the SNR. For all other throughput experiments, we fixed the nodes at a distance of 1 m. All experiments were performed in an RF-noisy office environment.

Figure 3-9 shows the SNR vs. BER plot for the 12 Mbps data rate. Each point in the graph represents the BER and the SNR values of 1000 temporally contiguous packets. We also plot the theoretical BER versus SNR values, which we computed using MATLAB's implementation of an optimum synchronizer, channel estimator, PSK/QAM demodulator, and Viterbi decoder. As seen from Figure 3-9, the SNR versus BER of the receiver follows the general exponential trend predicted by the theoretical model. Our measured performance is worse than the theoretical value by a few dB. For example, at a BER of 10^{-4}, the receiver's performance is worse by around 6 dB. Other data rates exhibited similar SNR vs. BER behavior.

We also measured Airblue's throughput at different packet sizes. Figure 3-10 plots the receiver throughput as a function of packet size, at bit-rates up to 24 Mbps. The measured SNR for the environment in which this experiment was run was 16

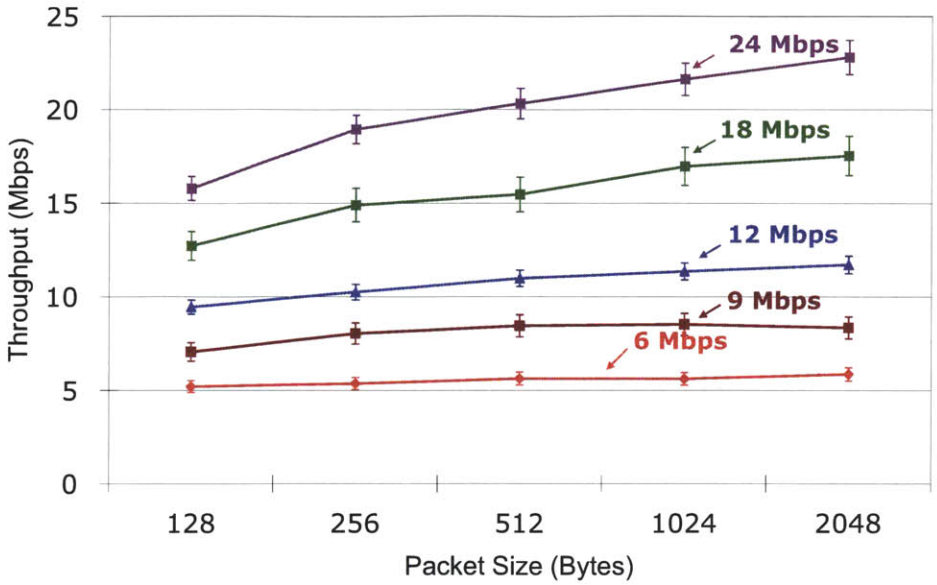


Figure 3-10: Throughput with different packet sizes.

dB. So, the BER at all bit-rates was less than 10^{-6} (Figure 3-9). For all bit-rates, we achieve the maximum throughput when transmitting large packets, as expected. The throughput decreases gradually for smaller packets, as the preamble and packet header overheads increase. Our prototype is able to meet various 802.11g timing specifications like turning around from receiving a packet to transmitting an ACK within 25 μs[2]. On average, the power consumption of the whole platform is 5 Watts throughout the experiment.

Program size: The total number of lines of Bluespec source code in our implementation is 17,250. Of this, 19.1% (3,288 lines) provides the arithmetic library used across the design, 40.9% (7,051 lines) implements various parameterized modules in the baseband PHY, 10.2% (1,761 lines) implements the device interface that controls the RF front-end, and 11.7% (2,022 lines) implements the MAC. The remaining 18.1% (3,128 lines) describes the top-level of our 802.11 design by instantiating the modules with the right parameters and connecting them together. Compiling our code with the Bluespec compiler results in 202,672 lines of RTL Verilog, which is more than 10

[2]802.11g requires that the ACK is transmitted within a slot time (9 μs) after the SIFS duration (16 μs).

Module	LoC	Logic Elms	Regs	RAM	DSP
FFT	1051	5577	8530	32	32
Receiver	4833	29622	22921	99	89
Synchronizer	1561	8958	7442	72	66
Channel Est.	762	4926	4709	25	23
Viterbi	1070	3361	2756	0	0
Demapper	276	9527	1646	0	0
Deinterleaver	97	2358	2071	0	0
Transmitter	1167	9348	7487	0	0
Cyclic Prefix	165	1064	2756	0	0
Pilot	95	2864	2805	0	0
Mapper	159	2201	1401	0	0
Interleaver	176	863	681	0	0
MAC	2022	2139	1693	0	0
Device I/F.	1761	3321	2756	0	0
System Total	17250	54320	43776	142	123

Figure 3-11: Lines of code (LoC) and synthesis results of our 802.11a/g transceiver: the physical implementation used on the FPGA was obtained using Synplicity Synplify Pro 9.4 for synthesis and Altera Quartus Fitter 8.0 for place and route. These results exclude support circuitry like the soft processor.

times the size of our source code.

Synthesis results: Synthesis results for our transceiver are presented in Figure 3-11. The transmitter is smaller than the receiver, because more complex algorithms are employed by the receiver to combat channel distortions. The synchronizer and the channel estimator are two of the most complex blocks in the receiver (loosely reflected by the lines of code), although neither is the largest block in terms of logic elements. These blocks would have used a lot more resources if there were no built-in multiplier units in the FPGA.

Overhead of latency-insensitive designs: We quantified the overhead of latency-insensitive (LI) designs due to extra buffering and control logic. For this purpose, we rewrote the *Channel Estimator* module in the baseband PHY in a latency-sensitive (LS) manner using registers only. The LS design operates iteratively over its input and output buffers, and assumes that adjacent modules sample those registers at

Hardware Resource	Latency-sensitive	Latency-insensitive
Logic Elements	5137	5438 (5.9%)
Storage Elements	2356	2429 (3.1%)

Figure 3-12: Comparing the cost of latency-sensitive vs. latency-insensitive implementations of the *Channel Estimator*. The numbers in parentheses show the overhead percentage incurred by the latency-insensitive design.

the appropriate clock cycles. We picked this block for evaluation because its ratio of computation elements to storage elements is representative of other blocks in our design. Figure 3-12 shows that the LI design requires 5.9% more logic elements for the control logic of the FIFOs and 3.1% more storage elements to keep track of the occupancy of the FIFOs than the LS design. We consider this overhead to be quite acceptable for the ease of adding new functionality.

3.5 Extending Airblue to Support Model-Based Simulation

While on-air operation is the end goal of Airblue, simulation is crucial to evaluate and debug the implementation. For example, recent wireless research [45, 90] proposes to modify the physical layer (PHY) of existing 802.11a/g to provide accurate bit-error rate (BER) estimates and to pass these estimates to the upper layers of the protocol stack, where they may be used to improve overall performance. There are two primary challenges to simulate such wireless systems for protocol evaluations.

First, validating a protocol often requires observation of events that occur infrequently. This is because many wireless protocols are able to recover data most of time even when the signal is severely corrupted. Therefore, the validation process is most interested in *rare* cases in which the data is corrupted. For example, the aforementioned proposal requires BER estimates that can predict BER as low as 10^{-9}, an operating point at which the vast majority of bits are received correctly. Therefore,

to achieve reliable measures for an algorithm that produces BER estimates, one needs to produce a statistically significant number of very uncommon events.

Second, it is difficult to obtain realistic traffic to test and debug the wireless systems. Broadcasting data on-air presents difficulty because it is nearly impossible to control the broadcast environment, rendering experiments irreproducible. On the other hand, generating synthetic traffic using set of mathematical channel models usually involves heavy use of high-complexity floating point operations and is best suited for software.

The first challenge implies that pure software simulator is not suitable because software simulation of detailed hardware is extremely slow; nodes in our simulation cluster process only a few kilobits per second. Parallel simulation across dozens of machines is not sufficient to produce enough of the rarer events for accurate characterization. Meanwhile, the second challenge suggests that accelerating the whole testbench on FPGA can also be problematic because the channel model is not amenable to hardware implementation. Therefore, the logical solution to the problem is FPGA co-simulation, which accelerates the simulation of the hardware pipeline using FPGAs but keeps the channel model implementation in software. The communication between the two are handled by a fast bi-directional link between the FPGA platform and the host PC.

As Airblue has already provided FPGA implementations of most baseband modules, it makes sense to reuse the available components to create a high-speed FPGA-accelerated simulator. Conceptually, we only need to replace the radio device interface by a synthetic traffic generator. In reality, this modification is in general not easy for conventional FPGA implementations because of the differences in performance characteristics, e.g., latency and throughput, and physical interfaces between the synthetic traffic generator and the radio device. On the other hand, Airblue's latency-insensitivity implementations and AWB's FPGA virtualization property make the task much simpler: We use the software synthetic traffic generator to implement the same LEAP's abstracted radio device interface. Even though the generator may have different latency and throughput when compared to the original Airblue's radio de-

vices, we can directly use the new implementation on our original baseband pipeline because the latency-insensitive property of these modules permits us to completely decouple the transmitter, receiver, and channel model.

3.5.1 Airblue Simulator Implementation

Our simulation environment consists of a Virtex-5 based ACP FPGA module [42] attached to a 1066 MHz front-side bus (FSB) and a quad core Xeon processor mounted to the same bus. This configuration provides a fast FIFO communication with bandwidth in excess of 700MB/s between the FPGA and the processor. Currently, we configure the FPGA to run the baseband pipeline at 35 MHz with the exception of the BER prediction unit, which runs at 60 MHz since it operates at per-bit granularity. This configuration allows our baseband pipeline to be capable of achieving the fastest transmission rate in 802.11g which is 54 Mbps in real-time on the FPGA.

For software channel, we implement an Additive White Gaussian Noise (AWGN) channel with a variable Signal-to-Noise-Ratio (SNR). To take advantage of the computation power of multi-core processors, our software channel implementation is multi-threaded.

Figure 3-13 shows the simulation speed of different rates achieved by our baseline 802.11 system with the software channel. We are able to achieve simulation speeds which are between 32.8% and 41.3% of the 802.11g line-rates. At the highest rate, we are able to achieve simulation speeds in excess of 20 Mbps. Experimental results show that all the simulation runs constantly use up only about 55 MB/s of the 700 MB/s available communication bandwidth between the FPGA and the processor, indicating that our software modules are the bottleneck of our system. Program analysis shows that computing noise values for the AWGN channel dominates our software time, even though the software is already multi-threaded to take advantage of the four available cores. Since noise generation alone was sufficient to saturate a quad core system, implementing the whole simulator in software will be too slow, thus validating our choice of FPGA-based co-simulation.

Modulation	Simulation Speed (Mb/s)
BPSK 1/2 (6 Mbps)	2.033 (33.9%)
BPSK 3/4 (9 Mbps)	2.953 (32.8%)
QPSK 1/2 (12 Mbps)	4.040 (33.7%)
QPSK 3/4 (18 Mbps)	6.036 (35.3%)
QAM-16 1/2 (24 Mbps)	8.483 (35.3%)
QAM-16 3/4 (36 Mbps)	12.725 (35.2%)
QAM-64 2/3 (48 Mbps)	15.960 (33.2%)
QAM-64 3/4 (54 Mbps)	22.244 (41.3%)

Figure 3-13: Simulation speeds of different rates. Numbers in parentheses are the ratios of the simulation speeds to the line-rate speeds of corresponding 802.11g rates

3.6 Discussion

While we show that 802.11g speed protocol can be implemented using Airblue, it is not amenable to implementing all wireless protocols. By nature, FPGA implementations trade some performance for reconfigurability. As a result, an FPGA implementation will not perform as well as an ASIC implementation. Although we are confident that Airblue can run recently deployed wireless protocols like 802.11n with some augmentation, proposed protocols operating above 10 Gbps will probably be out of reach of the FPGAs for the foreseeable future.

Airblue is a predominately hardware system. Designing high-performance hardware for a complicated function requires developers to manually extract the parallelism existing in the underlying algorithm and then express it in a parallel hardware language. Airblue does not free developers from this effort. Therefore, designing high-performance blocks in the PHY, like the Viterbi decoder, is still a challenge in Airblue. However, we believe that our architecture is considerably easier to modify than other experimental wireless systems. For less parallel blocks, like the MAC, modifications are more straightforward because they approach sequential programming.

3.7 Summary

Cross-layer protocols require new features and functions in various layers of the networking stack, a systematic way of passing additional information from lower layers to higher layers, and a fast way of controlling lower layers from higher layers. A development platform is not suitable for cross-layer protocol experimentation unless changes to the base protocols to implement such mechanisms can be made easily. In this chapter, we have discussed why the base protocols must be implemented in a *latency-insensitive* manner and must *pass control in a data-driven* manner to be modifiable by others. We have built a wireless experimentation platform called Airblue, which adheres to these design principles. In contrast, current platforms like SORA and WARP do not follow these design principles, and hence difficult to use for cross-layer experiments.

We believe that FPGAs represent an ideal platform for the development of new wireless protocols. First, a satisfactory FPGA implementation generally implies that a satisfactory ASIC implementation exists. Second, because of the infrequency of many interesting events associated with wireless transmission, high-speed simulation is needed to validate and characterize the implementation. To this end, we extend Airblue to also become a flexible and detailed co-simulation platform capable of detailed modelling of an OFDM baseband at speed close to the real line rate.

In the next chapter, using a relatively complex cross-layer protocol called SoftRate [91] as a case study , we will demonstrate that Airblue is easy to modify and that, when modified, it meets the performance requirements of current wireless protocols.

Chapter 4

Case Study: Evaluating SoftRate using Airblue

In the last chapter, we introduce Airblue and its implementation principles which are essential for modular refinements. In this chapter, we show how one can modify the MAC and PHY in Airblue to implement mechanisms that are useful for cross-layer protocols. The experiments in this chapter will demonstrate that Airblue provides both flexibility (comparable to a full software radio) and high performance (comparable to a hardware implementation). For our study, we targeted a relatively new and promising protocol proposed in the wireless community named SoftRate [91]. We chose this protocol because it has not been demonstrated in a high performance implementation, and because it covers a broad range of modifications required by cross-layer protocols. For completeness, we also discuss how to implement some cross-layer mechanisms that are not used to implement SoftRate in this chapter.

4.1 SoftRate and Its Requirements

SoftRate [91] is a rate-adaptation protocol proposed by Vutukuru et al. In SoftRate, when a receiver sends an acknowledgement packet to the transmitter, it also embeds the packet's bit-error rate (BER) estimate, i.e., the expected number of bits in the packet that are in error divided by the size of the packet. Upon receiving the acknowl-

edgement, the transmitter extracts the estimate and dynamically choose the optimal rate for future packet transmissions to this node.

SoftRate can be built on top of the SoftPHY interface [46, 45], which extends the Physical layer (PHY) of a wireless system to annotate each decoded data bit with an bit-error rate (BER) estimate and then send both information to the Media Access Control layer (MAC). Then, the SoftRate's MAC aggregates the per-bit estimates to produce per-packet estimates.

It is obvious that SoftRate is a cross-layer protocol: it requires the PHY to pass BER estimates to the MAC and the MAC to configure the PHY to adjust transmission rates. For a successful SoftRate implementation, there are several requirements. First, the BER estimates need to be very accurate. Indeed, SoftRate requires them to be able to predict BER as low as 10^{-7}. This is because a packet can have up to 10^4 bits. As a result, to be confident that a packet is without any bit error, its BER estimate need to be at the range of 10^{-5}. And to predict whether the packet can be sent at higher rate with no bit error, we need another 2 orders of magnitude margin to do so. As we have no mean to control the on-air environment to produce the desired BERs, our Airblue software channel extension will be useful to study the performance of various BER estimator implementations. Second, the platform needs to have enough bandwidth for the PHY to pass the BER estimates to the MAC. In this case, each decoded bit is associated with the 9-bit BER estimate, effectively requiring 10 times the bandwidth. Third, the transmitter needs to receive the acknowledgement within the time limit specified by the 802.11g standard. Although we can relax the latency requirement, it is not desirable because this increases the idle time of the channel as no node is supposed to use the shared channel while the transmitter is waiting for the acknowledgement.

In the remaining of the chapter, we show how we obtain a satisfactory implementation of SoftRate using Airblue. We started with studying two implementations of the BER estimators based on two different algorithms in Section 4.2. We compare their hardware complexities in the context of FPGA resource usage as well as their accuracies, in the context of both the BER prediction and the rate adaptation, under

AWGN channel. Then, in Section 4.3, we discuss the modifications to other modules in the system in order to obtain real-time on-air capable implementation. After that, we demonstrate how to implement additional cross-layer mechanisms that are useful to implement other cross-layer protocols in Section 4.4. Finally, we conclude the chapter in Section 4.5.

4.2 Estimating BER

As mentioned earlier, SoftRate requires the PHY to provide BER estimates to the MAC. However, it is difficult to estimate the BER of a channel accurately, because the receiver normally does not know in advance the content of the transmitted data. Furthermore, the channel behavior itself may be highly variable, even across a single packet, and must be measured frequently to obtain accurate BER estimates. Fortunately, the SoftPHY abstraction offers a solution to the problem of fine-grained BER estimations. SoftPHY makes use of a soft-decision convolutional-code decoder to export a confidence metric, the log-likelihood ratio (LLR) of a bit being one or zero, up the networking stack. While this work has shown that SoftPHY is able to produce high quality BER estimates, it has been evaluated only in software and does not meet the throughput (54-150 Mbps) or the latency (25 μs) requirements of high-speed wireless standards such as 802.11a/g/n. For SoftPHY to be useful, it must be implemented efficiently in hardware while meeting these performance targets.

Soft-decision convolutional-code decoders are commonly used as a kernel for decoding turbo codes [9], and numerous hardware implementations [55, 6, 10, 2, 54] have been optimized for this purpose. These implementations are based on either the BCJR algorithm [4] or the SOVA algorithm [35]. The former usually provides better decoding performance but involves more computation and more complex hardware. To reduce hardware complexity, all these implementations ignore the signal-to-noise ratio (SNR) during the calculation of LLR. While this optimization does affect the performance of turbo codes because they require the LLR outputs only to maintain their relative ordering, it is unclear that the same optimization will be as effective

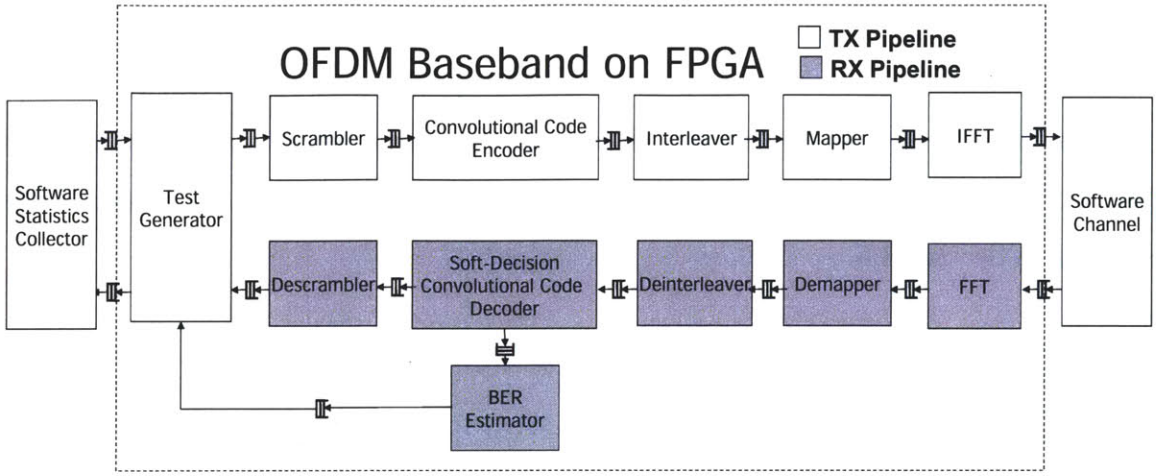


Figure 4-1: Components required to validate a BER estimator in a co-simulation environment.

to SoftPHY BER estimations which need to also take into account the magnitude of these values.

To study whether these implementations can be used for BER estimations, we implemented SoftPHY based on both BCJR [6] and SOVA [10]. Then, we empirically evaluated and characterized our designs through simulations of SoftPHY in the context of an 802.11-like OFDM baseband processor shown in Figure 4-1.

4.2.1 Convolutional Code Processing

The accuracy of our BER estimation is, in part, determined by the performance of the baseband processor in which it operates. For the sake of completeness, we will briefly describe again the baseband components most relevant to BER estimation in the following. Chapter 2 contains a more complete description of the OFDM baseband processing.

Convolutional encoder: A convolutional encoder is a shift register of $k - m$ bits where k and m are the constraint length and input symbol bit-length respectively. At each time step, an encoder with coding rate of m/n ($n > m$) generates an n-bit output according to n generator polynomials, each specifying the bits in the shift register to be "XORed" to generate an output bit. In our experiment, we use the convolutional code of 802.11a which has constraint length of 7 and code rate of 1/2.

Soft-Decision Convolutional Code Decoder: A soft convolutional decoder produces at its output a decision bit $\hat{b_i}$ and an log-likelihood ratio (LLR) denoting the confidence that the decision is correct with the following definition:

$$LLR_{dec}(i) = log\frac{P[\hat{b_i} = b_i|y]}{P[\hat{b_i} \neq b_i|y]} \tag{4.1}$$

which is the ratio of the probability that the bit is correctly decoded ($\hat{b_i} = b_i$) to the probability that the bit is incorrectly decoded ($\hat{b_i} \neq b_i$). Given y, the decoder determines the most likely state sequence of the shift register from the encoder that would generate y. There are two common algorithms to decode convolutional code and output LLRs: the SOVA algorithm [35] and the BCJR algorithm [4]. We implemented both, which are discussed in details in Section 4.2.3, for a proper hardware evaluation. Next, we discuss the demapper that provides the inputs of the decoder and its hardware implementation.

Demapper: The convolutional code demapper maps each subcarrier's phase and amplitude to a particular set of bits, based on the transmitter modulation scheme. Since these values maybe distorted by the channel, the demapper also assigns a LLR to each demapped bit with the following definition.

$$LLR_{demap}(i) = log\frac{P[b_i = 1|r[k]]}{P[b_i = 0|r[k]]} \tag{4.2}$$

which is the ratio of the probability that the i-th decoded bit is 1 to the probability that the decoded bit is 0 given the received symbol $r[k]$ at time k that contains bit i.

A good approximation [80] of this LLR under a flat-fading Additive White Gaussian Noise (AWGN) channel can be obtained with the following equation.

$$LLR_{demap}(i) = \frac{E_s}{N_0} \times S_{modulation} \times R_{dist}(i) \tag{4.3}$$

which is the ratio ($R_{dist}(i)$) between the Euclidean distance of the received symbol to the closest 1 and the distance to the closest 0, multiplied by the signal-to-noise-ratio ($\frac{E_s}{N_0}$) and a constant depending on the modulation scheme ($S_{modulation}$).

We base our demapper on Tosato et al. [87], who further optimize the calculation of $R_{dist}(i)$ by eliminating multiplications and divisions. If $\frac{E_s}{N_0}$ remains roughly the same for all data subcarriers and across the packet transmission, further optimization can be made by ignoring $\frac{E_s}{N_0}$ and $S_{modulation}$ due to the fact that the bit-decoding decisions are determined by the relative ordering of the terms in the convolutional decoding computation instead of their magnitudes. This optimization allows the decoder to achieve the same decode performance with reduced bit-width (23-28 bits $\rightarrow$ 3-8 bits), which helps significantly reduce the area of the decoder. Unfortunately, the magnitude of the computation is important when estimating the BER.

4.2.2 BER Estimation

Our BER estimator takes per-bit LLR estimates from the soft decision decoder and translates them into per-bit BER estimates. These estimates may be processed before they are passed up to higher levels, for example by calculating the packet BER.

From equation 4.1, the LLR estimate can be converted to a per-bit BER with the following equation.

$$BER_{bit} = \frac{1}{1 + e^{LLR_{dec}}} \tag{4.4}$$

Unfortunately, the LLR estimates produced by either BCJR or SOVA are only approximations of the true LLR. This imprecision has two causes: first, the SNR and the modulation factors (as shown in equation Section 4.3) are ignored when the hardware demapper generates the inputs for the decoder; second, the input values are interpreted using different scales by the hardware BCJR and SOVA. We study the impact of these input scalings to a LLR estimate output from both algorithms [4, 35] and find this estimate can be converted to to the true LLR ($\hat{LLR}_{dec}$) with the following equation.

$$\hat{LLR}_{dec} = \frac{E_s}{N_0} \times S_{modulation} \times S_{dec} \times LLR_{dec} \tag{4.5}$$

where $\frac{E_s}{N_0}$ is the SNR, $S_{modulation}$ is a constant scaling factor determined by the mod-

ulation scheme and S_{dec} is another scaling factor determined by the decoder.

One way to implement the per-bit BER estimator is to mathematically calculate the precise value for each scaling factor and then adjust the LLR according to equation 4.5. After that, the per-bit BER can be obtained by using a lookup table generated following equation 4.4. While the last two factors can be computed statically, the SNR needs to be estimated at run-time.

Instead of implementing an SNR estimator, we believe that a pre-computed constant for SNR is sufficient. We observe: 1) we only need the BER prediction to be accurate up to the order of 10^{-7} because a maximum size of a packet is usually in the order of 10^4 bits. While the order of 10^{-5} is sufficient for checking packet errors, extra margin can help rate adaptation protocols like SoftRate [91] to identify potential of sending packets at higher rate; 2) the range of SNR over which a modulation's BER drops from 10^{-1} to 10^{-7} is only a few dB [23]. Therefore, we can pick an appropriate SNR constant, i.e., a value in the middle of the SNR range mentioned above for each modulation and still get reasonably accurate BER estimates. This proposal will slightly underestimate the BER if the actual SNR is lower than the chosen middle value and overestimate the BER if the SNR is higher. With this simplification, we can implement a BER estimator as a two-level lookup. Given an LLR output from the decoder and the modulation scheme, we look up the right table and obtain the BER.

4.2.3 Soft Decision Decoder Architecture

A convolutional encoder is implemented with a shift register. At each time step, it shifts in an input bit, transits to the next state, and produces multiple bits as an output based on the transition. By observing only these outputs, as determined by the demapper, a decoder attempts to determine the most likely state transitions of the encoder. In contrast to hard decision decoders, which output a single decision bit, soft decision decoders produce at their output a decision bit and an LLR denoting the confidence that the decision is correct. Both SOVA and BCJR require minor augmentation to calculate these ratios.

Theoretical work [57] has shown that BCJR and SOVA are deeply related: both SOVA and BCJR decode the data by constructing one or more *trellises*, directed graphs comprised of all the state transitions across all time steps. Each column in a trellis represents all the possible state of the shift register in a particular time step. For example, there will be 2^n nodes in a column if the size of the shift register is n bits. Two nodes are connected with a directed edge if it is possible for the encoder to reach one from the other by way of a single input. Each node is associated with a value called the path metric. Although path metrics have different meanings in BCJR and SOVA, they generally track how likely it is that the encoder was in a particular state at a particular time.

Two kernels are used to calculate path metrics: the branch metric unit (BMU) and the path metric unit (PMU). At each time step, the BMU produces a branch metric for each possible transition by calculating the distance between the observed received output and the expected output of that transition. This distance constitutes an error term: if it is large, then the output associated with the distance is not likely. Then, the PMU calculates the new path metric for each transition by combining the corresponding branch metric with the path metric of the source node from the previous timestep. As both SOVA and BCJR use BMU and PMU, the designs of these two components are shared. The PMU is parameterized in terms of path permutation, which differs between the forward and backward trellis paths of BCJR, and the Add-Compare-Select (ACS) units, which can be different between SOVA and BCJR. The BMU is identical in SOVA and BCJR.

SOVA and BCJR differ in the way they use path metrics to determine the directed edges in the trellis. SOVA attempts to determines the most likely state sequence along a period of time. SOVA requires the PMU to provide the path metrics and their corresponding previous states, i.e., survivor states, at each time step. Using this information, it constructs a sliding traceback window that stores columns of survivor states it received most recently. For each window, SOVA performs a traceback which starts from the node with the smallest path metric for the current time step, and then iteratively follows the survivor state at each earlier time step until it reaches a node

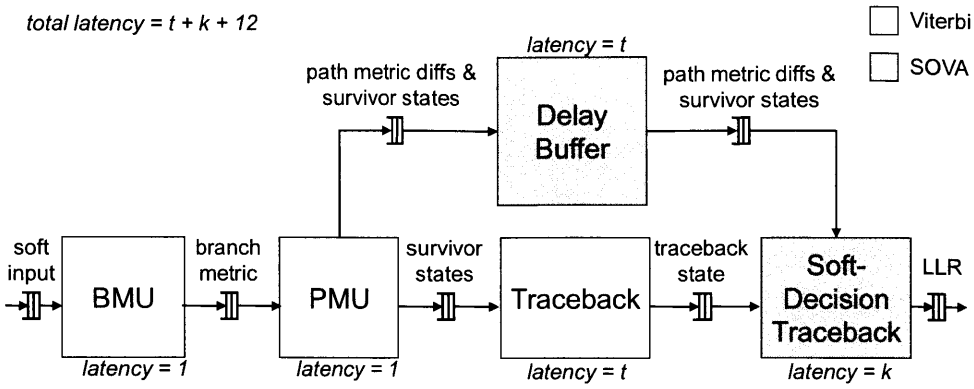


Figure 4-2: SOVA pipeline: Blocks in white exist in hard-output Viterbi while blocks in grey are SOVA exclusive. Text in italic describes the latency of each block.

belong to the earliest time step in the window. This node is then used to determine the original input to the encoder at that time step.

On the other hand, BCJR seeks to compute the most likely state of the convolutional encoder at each timestep. Given the complete set of encoder outputs, BCJR first calculates the path metric for each state at each timestep moving in a forward direction (α_i) and then computes the path metric for each state in each timestep in the reverse direction (β_i), determining the new path metrics by summing the branch metric - path metric product of incoming trellis edges. Finally, the forward and reverse probabilities for each timestep are combined with the branch transition metric (γ_i) to produce a likelihood for each state at each timestep. The most likely state at each timestep determines the most likely bit input into the convolutional encoder at that timestep.

In the remaining of the section, we discuss the architectures and the implementation challenges of SOVA and BCJR respectively.

SOVA

Figure 4-2 shows the structure of our hardware SOVA pipeline, which is based on the one shown in [10]. The pipeline consists of a BMU, a PMU, a delay buffer and two traceback units, all connected by FIFOs.

The two traceback units construct two traceback windows to find the most likely

state at each timestep. The results from the first are used as better initial estimates for the second. The second traceback unit also outputs the LLRs. It does so by also keeping track of soft decisions, one for each timestep. Each soft decision represents the confidence of the decoded bit at that timestep. The second traceback unit performs two simultaneous tracebacks, tracking the best and the second best paths, starting with the output state received from the first traceback unit. At each step of the traceback, the states from the two paths are compared. If the two states output different hard decode decisions and the difference of the two path metrics is smaller than the corresponding soft decision, this decision is updated with this smaller value.

The total latency of our SOVA implementation is $l + k + 12$ cycles. l and k are the traceback lengths of the first traceback unit and the second traceback unit respectively. Each BMU and PMU adds an extra cycle of latency. Each FIFO has 2 elements and thus adds at most 2 cycles to the total latency. Therefore, 5 FIFOs add another 10 cycles. If the l and k are both 64, the total latency will be 140 cycles. As our design runs at 60 MHz at least, the latency is no more than 2.3 μs, which implies it can be used in protocols with tight latency bound (25 μs for 802.11a/g).

BCJR

The major difficulty in implementing BCJR lies in the calculation of the backward path metrics. Waiting for an entire frame of data before beginning computation is unacceptable, both in terms of the latency of processing and in terms of storage requirements. To avoid these issues, we approximate BCJR by operating on sliding blocks of reversed data, the SW-BCJR [6]. Thus, we reverse each block of n data, and determine the backward path metrics of that block in isolation. By making n small, we reduce the latency of the algorithm and reduce storage requirements, at the cost of some accuracy.

However, blocking alone is not enough. In order to process the backwards path for a block p, BCJR must know the final path metric for the succeeding block $p+1$. Unfortunately, this information can only be determined by calculating the reverse path metrics on the remainder of the packet, which we have not yet received. To provide

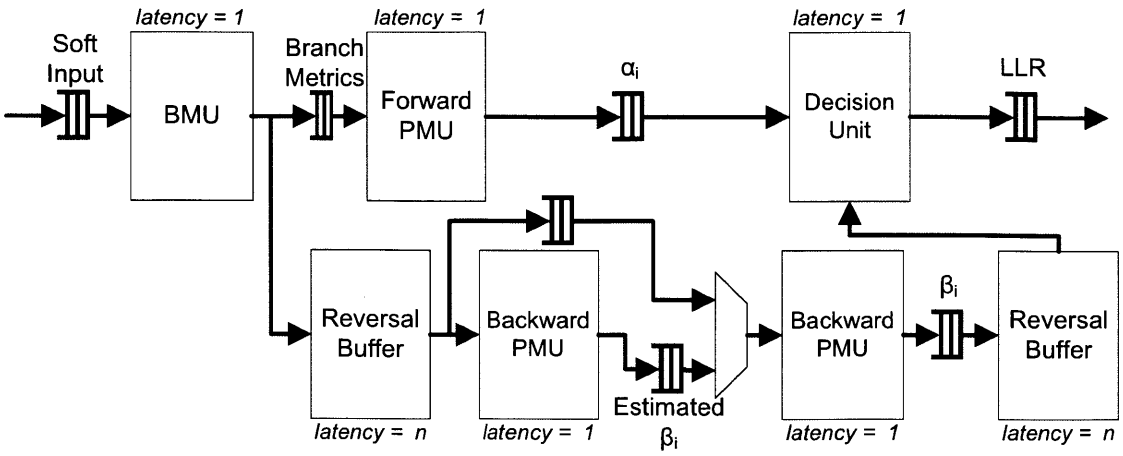
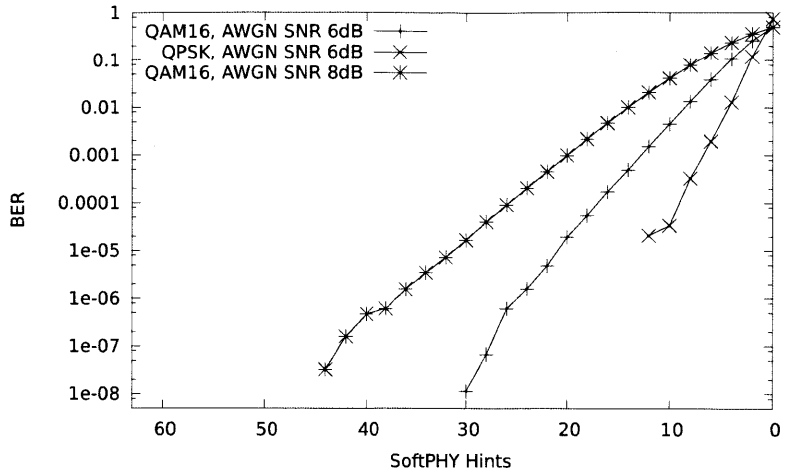


Figure 4-3: BCJR pipeline.

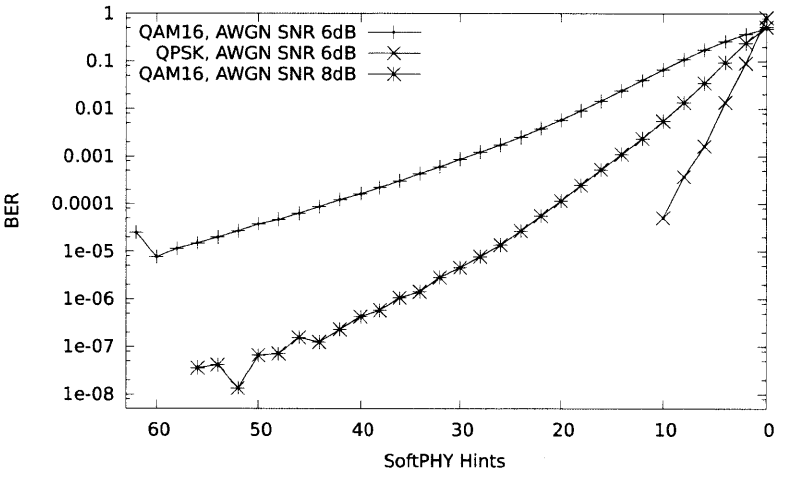
an estimated final path metric for block $p+1$ when calculating block p, we perform a provisional path metric calculation on block $p+1$. Of course, this computation also has uncertain start state, but in this case we use a default "uncertain" state as the initial metric. This configuration shows reasonable performance if block size n is sufficiently large (larger than 32).

Figure 4-3 shows our streaming BCJR pipeline. Our implementation consists of three major streaming kernels, PMU, BMU, and decision unit, which selects the most likely input bit. In addition to the extra PMU needed for calculating provisional backward path metrics, the backward path also needs a pair of memories to reverse and unreverse blocks. The reversal buffers that we use to re-orient the data frames in the backwards path are based on dual-ported SRAMs. They are streaming, with a throughput of one data per cycle and a latency equal to their size. The pair of reversal buffers and the large FIFO required to cover the latency of the provisional PMU represent a substantial overhead in our architecture. Adding SoftPHY functionality to the architecture is simple: we modify the decision unit to choose both the most like '1' state and the most likely '0' state, subtracting the path metrics of the two states to obtain the LLR. This approach adds only a single subtracter to the pipeline and has no impact on timing.

The latency of BCJR is dominated by the latency of the reversal buffer units, which must buffer an entire block before emitting data. With a reversal buffer of size



(a) BCJR



(b) SOVA

Figure 4-4: BER v. LLR Hints, across different modulation schemes and noise levels

n the latency of BCJR is $2n + 7$, with pipeline and FIFO latency causing the extra constant term. At 60 MHz with a block size of 64 this corresponds to a latency of 135 cycles, or 2.2μs, which is comparable to the latency of SOVA with traceback length set to 64.

4.2.4 Evaluation

Using Airblue, we evaluate different aspects of our SoftPHY implementations. First, we study the relationship between the LLR values produced by our hardware de-

coders and the actual BERs. Then, we evaluate the accuracy of our per-packet BER estimator in the context of SoftRate. Finally, we compare the hardware complexities of our SOVA and BCJR decoders.

Relationship between LLRs and Per-Bit BERs

As our Airblue implementations are approximate algorithms, we must show that the LLR values produced by the hardware decoders correspond well to the LLR suggested by theory. To determine the relationship between these LLR values and the BERs, we simulated the transmission of trillions (10^{12}) of bits on the FPGA. Several resulting curves are shown in 4-4. Both BCJR and SOVA are able to produce LLRs showing the log-linear relationship with BERs as suggested by the equation 4.4 in Section 4.2.2. As expected, the slopes of the curves vary with SNR, modulation, and decoding algorithm, validating the 3 scaling factors we proposed in equation 4.5. As a result, we can use these curves to determine the values of these scaling factors and to generate lookup tables for our per-bit BER estimator.

It is important that our implementations are able to produce LLRs that cover a wide range of BERs (i.e., 10^{-7} to 10^{-1}). High per-bit BERs (10^{-2} or above) can predict which bits in the packet are erroneous while low BERs (10^{-7} to 10^{-5}) can predict how likely the whole packet has no error. Although both SOVA and BCJR can produce LLRs that can predict BERs lower than 10^{-7} for some SNRs, BCJR can produce them at a wider range of SNRs than SOVA.

Accuracy of Per-Packet BER Estimates

Per-packet BER (PBER) can be obtained simply by calculating the arithmetic mean of the per-bit BER estimates in a packet. This measure is useful as means of condensing the per-bit BER for communication with higher level protocols. Figure 4-5 shows the graph plotting the actual PBERs against the predicted PBERs. The predicted PBERs are reasonably clustered around the ideal line, except for high BERs (10^{-1} or above), where there is slight underestimation. These underestimations are a result of the constant SNR adjustment we apply to the decoder's LLR outputs, as discussed

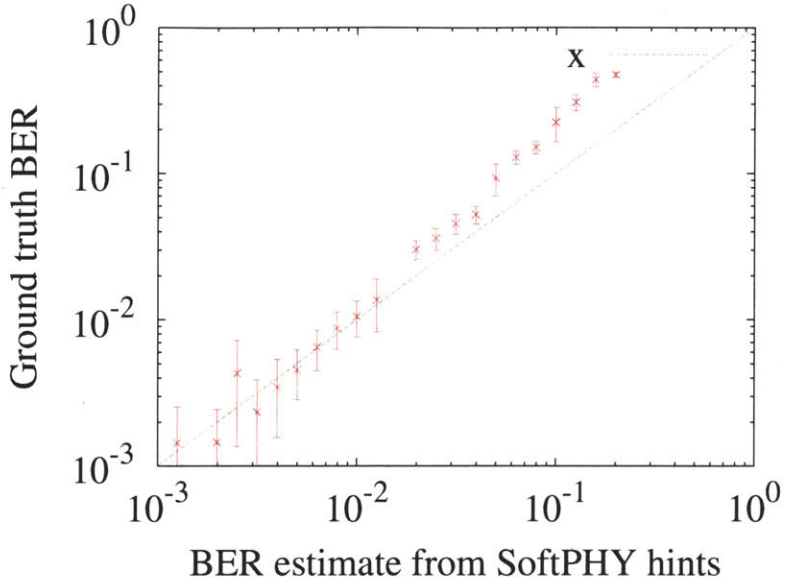


Figure 4-5: Actual PBER v. Predicted PBER (Rate = QAM16 1/2, Channel = AWGN with varying SNR, Packet Size = 1704 bits). The line represents the ideal case when Actual PBER = Predicted PBER. Each cross with the error bar represents the average of the actual PBERs for that particular predicted PBER value with a standard deviation of uncertainty.

in Section 4.2.2.

To further test the accuracy of our PBER calculations, we implement SoftRate [91] in Airblue. SoftRate is a recently proposed MAC protocol which makes use of PBERs to better decide rates at which packets can be transmitted. If the calculated PBER at the current rate is outside of a pre-computed range (for the ARQ link layer protocol, the range is between 10^{-7} and 10^{-5}), then SoftRate will immediately adjust the future transmission rate up or down accordingly.

In this experiment, the transmitter MAC observes the predicted PBERs emitted by the receiver estimator and adjusts the rate of the future packets, approximating a full transceiver implementation, in which the packet BER estimate would be attached to an ARQ acknowledgement message. We use a pseudo-random noise model which allows us to test multiple packet transmissions at various rates with the same noise and fading across time. We consider the optimal rate to be the highest rate at which a packet would be successfully received with no errors: a rate picked by SoftRate is overselected (underselected) if this rate is higher (lower) than the optimal

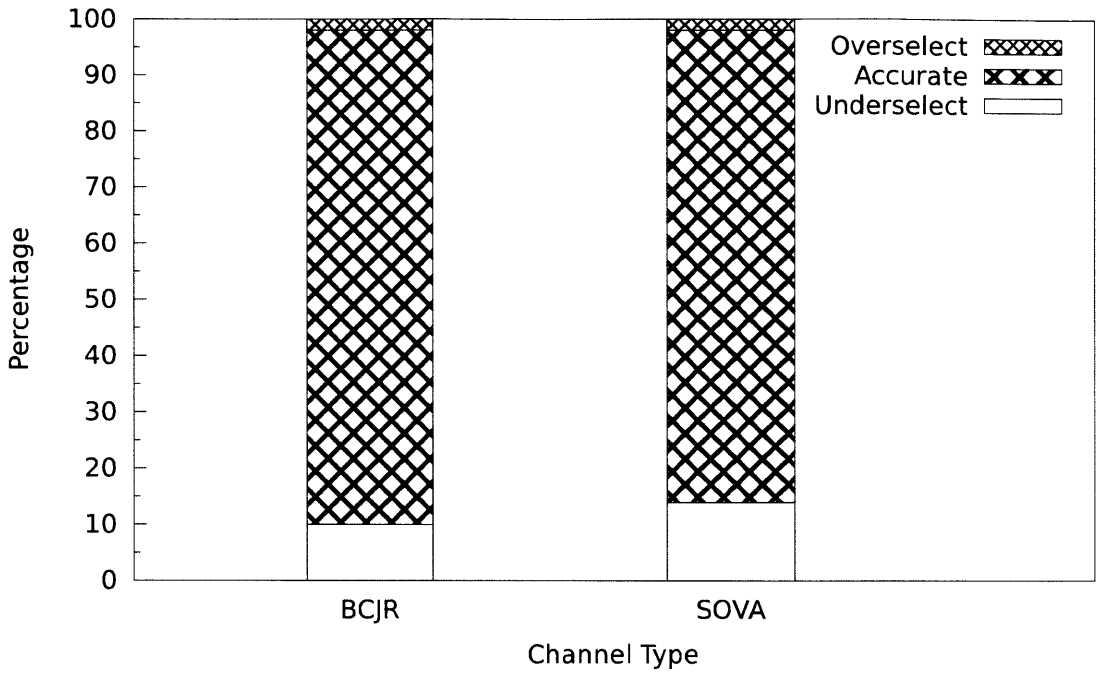


Figure 4-6: Performance of SoftRate MAC under 20 Hz fading channel with 10 dB AWGN.

rate. Figure 4-6 shows the performance of our SoftRate implementations with BCJR and SOVA under a 20 Hz fading channel with 10 dB AWGN. Both implementations are able to pick the optimal rate over 80% of the time, suggesting that both produce sufficiently accurate PBERs. As expected, SOVA picks the optimal rate less frequently than BCJR by a small margin: SOVA underselects the rate 4% more often than BCJR while both overselect 2% of the time.

Implementation Complexity

Experimental evaluation suggests that BCJR produces superior BER estimates, both per-bit and per-packet. However, this production comes at a high implementation cost. Figure 4-7 compares the synthesis results of the BCJR and the SOVA decoders using Synplify Pro 2010.09 targeting the Virtex 5 LX330T at 60 MHz. As a baseline, we also show the synthesis results for a Viterbi decoder implementation, as is typically used in commodity 802.11a/g baseband pipelines. We target a processing speed of 60 Mbps, since the maximum line rate of 802.11a/g is rate of 54 Mbps and our decoders

Module	LUTs	Registers
BCJR	**32936**	**38420**
Soft Decision Unit	6561	822
Initial Rev. Buf.	804	2608
Final Rev. Buf.	8651	30048
Path Metric Unit	4672	0
Branch Metric Unit	63	41
SOVA	**15114**	**15168**
Soft TU	13456	13402
Soft Path Detect	7362	4706
Viterbi	**7569**	**4538**
Traceback Unit	5144	3927

Figure 4-7: Synthesis Results of BCJR, SOVA and Viterbi. SOVA is about half the size of BCJR.

are capable of emitting one bit per cycle. Although our designs are optimized to use FPGA primitives like Block RAM, for the purpose of comparison we force the tools to synthesize all storage elements to register.

BCJR is about twice the size of SOVA, primarily due to the three path metric units used by BCJR and its larger buffering requirements. Although BCJR uses fewer registers, this is because it uses large amounts of BRAM. Meanwhile, SOVA itself is about twice the size of Viterbi. The area of both SOVA and BCJR can be reduced by shrinking the length of the backward analysis. In our current implementation, we use a backward path length of 64 for SOVA and a block length of 64 for BCJR. We find that increasing these values provides no performance improvement.

Accuracy of Airblue Modelling

All models, including those constructed using Airblue, lose some fidelity as compared to a real implementation. In the case of our Airblue experiments, our model of the wireless baseband is extremely detailed and accurate: it has been used to build high quality radio transceivers in Airblue. However, the channel models used by Airblue are certainly approximations of a real wireless channel, and the on-air capabilities of the modules that we have introduced in this study are unknown.

Because we do not have an on-air implementation of SoftPHY or SoftRate, the best comparison that we can make is against previously published [91]. Although the original SoftPHY results were trace-driven, they were based on on-air data collection and provide at least some basis for comparison. Airblue based-simulation suggests an accuracy rate of 85% for the SoftRate protocol, while the original paper achieved only a 75% accuracy, a differential of 16%. Our hardware model, being an approximation, should have intuitively *underperformed* the ideal software implementation originally proposed. There are likely three contributing error terms in Airblue simulation. First, our channel model is relatively simple. Second, we took steps to compensate for SNR variability in our SoftPHY implementation, while the original implementation ignored these issues. Third, we did not model channel estimation or synchronization in the receiver. These three factors would serve to increase the apparent performance of SoftRate. Ultimately, we view the discrepancy between the two experiments as acceptable: the offered performance gain of SoftRate is high, around 2x to 4x depending on the base of comparison.

4.2.5 Verdict: BCRJ is More Accurate But SOVA is Less Expensive

We used Airblue to evaluate two hardware implementations of SoftPHY, a recently proposed protocol. Although the BCJR implementation of SoftPHY outperformed the SOVA implementation, the latter performed acceptably well, and at less than 50% of the area of the former. Generally speaking, the hardware implementations were quite successful at predicting BER with what we believe is an acceptable hardware cost (around 10% increase in the size of a transceiver), indicating that SoftPHY is a competitive augmentation to future wireless chips and protocols.

Section	Experiment	Results
Section 4.3.1	Sending per-packet feedback.	Modified to generate link-layer ACK with channel quality feedback in 15.71 μs, meeting 802.11a timing requirement, in 20 lines of code.
Section 4.3.2	Computing and exporting SoftPHY hints.	Replaced the Viterbi decoder with the soft output BCJR decoder that computes SoftPHY hints. This modification increases the receiver pipeline processing latency from 8.28 μs to 9.91 μs, but does not affect throughput. The latency-insensitive nature of the design ensures that the modifications are limited to the modules that compute and export SoftPHY hints.

Figure 4-8: Experiments to implement SoftRate with Airblue for on-air experimentations. Most timing results presented are within the typical ranges expected in wireless standards like 802.11, making Airblue suitable for running realistic experiments.

4.3 Implementing SoftRate in Airblue

In the last section, we study two implementations of a BER estimator. In this section, we how we modify the baseline Airblue implementation to support SoftRate. To show the flexibility of Airblue, we decide to pick BCJR as the default BER estimator because of it is substantially different from the Viterbi in terms of architecture when compared to SOVA. The main results in this section are summarized in Figure 4-8.

4.3.1 Sending Per-packet Feedback

We modify Airblue to send per-packet feedback from the receiver PHY to the receiver MAC, and subsequently to the MAC layer at another node via the link-layer ACK frame. This mechanism is useful for a variety of cross-layer MAC protocols, e.g., to send SNR or BER estimates for bit rate adaptation [40, 91]. We consider the specific example of sending the sum of SoftPHY hints as channel quality feedback, but the results described here broadly apply to sending other types of feedback as well.

The streaming interface between the PHY and the MAC delivers SoftPHY hints and data bits to the MAC as they are decoded at the PHY. We add a new module to

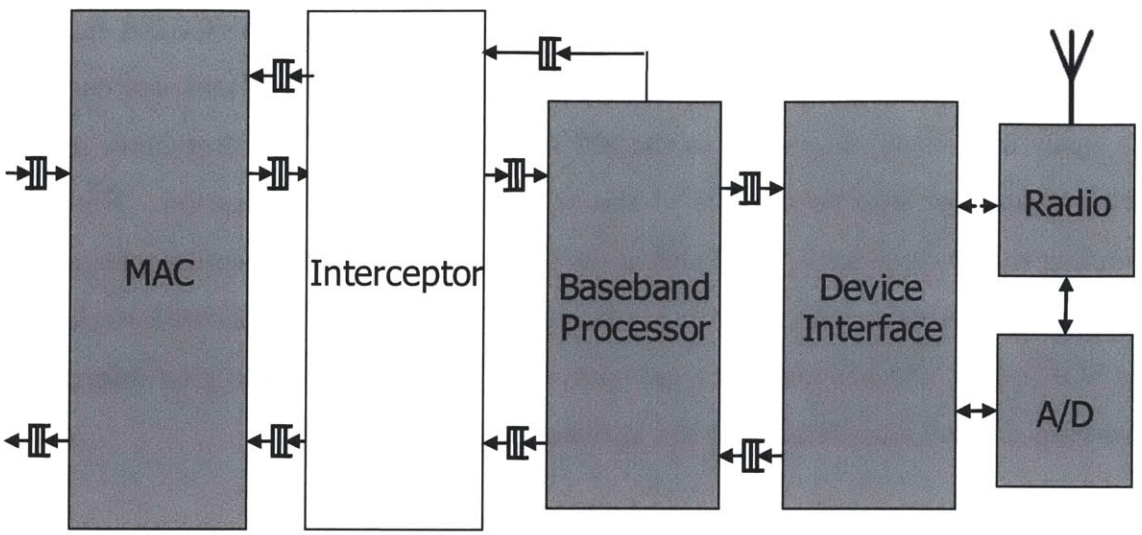


Figure 4-9: An interceptor module in the MAC to enable easy modifications to the standard CSMA MAC.

the MAC, called the *interceptor*, which sits between the baseband PHY and the CSMA MAC, as shown in Figure 4-9. The interceptor provides a composable functionality to the MAC, by augmenting its function without requiring modification to the MAC itself. In this case, the interceptor snoops the data exchanged between the PHY and the MAC, computing the running sum of the per-bit SoftPHY hints exposed by the PHY in the same clock cycle that they arrive at the MAC. The final feedback is ready 9.91 μs after the packet transmission completes, the *receiver pipeline processing latency* when computing SoftPHY hints (see Section 4.3.2).

After the feedback is ready, it takes the MAC a further 3 μs to check the CRC on the packet and decide whether to send the link-layer ACK or not. If the MAC decides to send an ACK frame, the interceptor modifies the initial ACK frame, embedding feedback in the payload of the ACK as the ACK streams through the interceptor. This operation has no impact on the ACK latency. Finally, it takes the PHY transmitter another 2.8 μs to transmit the first OFDM symbol over the air after it receives the transmit request, for a total of 15.71 μs (9.91 + 3 + 2.8) to send a link-layer ACK embedding feedback. To put this delay in perspective, 802.11a stipulates that the link-layer ACK must be transmitted within a slot time (9 μs) after the SIFS duration (16 μs), and our implementation comfortably meets this requirement.

All of the changes described above to the interceptor were performed in under 20 lines of code. The latency-insensitive nature of Airblue allows protocol designers to easily implement extensions to the 802.11 link layer using the interceptor module without delving into the details of the original MAC implementation. Note that sending variable per-packet feedback in the link-layer ACK while meeting microsecond timing constraints is practically impossible to do in software-only network stacks such as SORA [86], which require *at least* many tens of microseconds to pass information between the software MAC and the hardware radio front-end.

4.3.2 Computing SoftPHY Hints

As mentioned before, we replace the hard output Viterbi decoder with the BCJR decoder discussed in the previous section, as shown in Figure 4-10. This experiment illustrates the ease with which one can modify the processing pipeline of a layer in Airblue to add new functionality. We implemented the new decoder in 815 lines of new code, reusing several components from our Viterbi implementation. Because BCJR examines multiple backwards paths through the packet, the BCJR decoder has a longer pipeline latency than the Viterbi decoder, increasing our receiver pipeline processing latency from 8.28 μs to 9.91 μs, equivalent to an addition of 98 cycles at the 60 MHz clock. However, this large change in latency of decoder did not affect the correctness of any other module in the pipeline, due to the latency-insensitive nature of our design. Although the processing latency increases due to SoftPHY computation, the throughput of the PHY pipeline is unaffected because both decoders are capable of decoding 1 bit per cycle.

To export SoftPHY hints from the PHY to the MAC, we pass the hints along with the data by simply extending the data types of the interfaces between the modules downstream to the decoder to hold a 9-bit SoftPHY hint in addition to the data bit. This implementation required changing 132 lines of code in the *Header Decoder* and *Descrambler*, as shown in Figure 4-10. Note that our implementation requires the communication bandwidth between the PHY and the MAC to be widened from 8 bits to 80 bits per cycle at 25 MHz. This is both reasonable and easy to implement

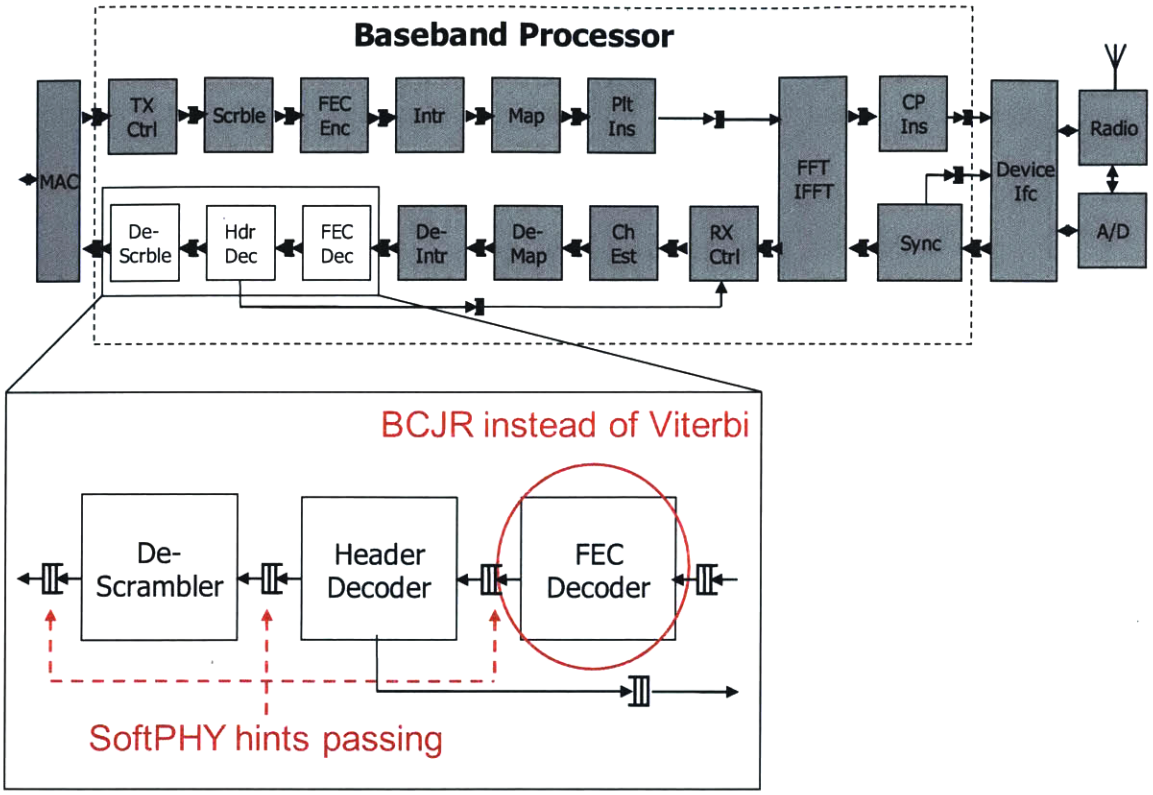


Figure 4-10: Modifications to the baseband PHY pipeline to compute and export SoftPHY hints.

because both the PHY and the MAC are implemented in hardware, which gives us flexibility to adjust the communication width to meet the bandwidth requirement. Had the MAC been implemented in software, this modification may have been been impossible if there were insufficient communication bandwidth between the MAC and the PHY.

4.4 Implementing Other User Cross-Layer Mechanisms in Airblue

In the previous section, we have shown that Airblue is capable of implementing SoftRate while meeting all the performance requirements. In this section, we demonstrate how Airblue can be used to implement another very useful cross-layer mechanisms which the results are summarized in Figure 4-11.

Section	Experiment	Results
Section 4.4.1	Decoding MAC header during reception.	MAC-layer information starts streaming up to the MAC in 9.91 μs after transmission. Implemented in 43 lines of code.
Section 4.4.2	Runtime reconfigurations through interrupts.	The MAC can interrupt and reconfigure the receiver pipeline in 4.67 μs. Implemented in 115 lines of code.

Figure 4-11: Experiments to implement cross-layer mechanisms with Airblue. Similar to the SoftRate experiments, most timing results presented are within the typical ranges expected in wireless standards like 802.11. Moreover, all modifications are less than 1% of the project's code base, signifying the flexibility of the platform.

4.4.1 Decoding MAC Header During Packet Reception

Some cross-layer protocols like ZigZag [30] and Conflict Maps (CMAP) [92] depend on knowing the MAC-layer source and destination addresses before the packet reception completes. We now illustrate the usefulness of our streaming MAC-PHY interface in exchanging such information quickly between the two layers.

802.11 packets typically consist of two headers, one for the MAC and one for the PHY. While receiving samples, the PHY must first decode its header to know which modulation and coding to use to decode the MAC header, then reconfigure the pipeline accordingly before the MAC header can be decoded. To avoid this additional delay of reconfiguring the pipeline before MAC-layer information can be passed up, we modify the packet format in our implementation to send the time-critical portions of the MAC header at the lowest bit rate just after the PHY header,[1] and the rest of the MAC header at the higher payload rate. Note that sending the MAC-layer information at the lowest rate has the beneficial effect of increasing its reliability, thereby improving protocol performance. We implemented this mechanism by modifying just 43 lines of code in the *TX Controller* and *Header Decoder* modules in Figure 3-7 and the *Interceptor* module in Figure 4-9. In the new implementation, the streaming interface at the PHY now passes up this portion of the MAC header in

[1]We transmit the 8 LSBs of the source and destination MAC addresses and some parity bits in 2 OFDM symbols at the 6 Mbps base rate.

the same clock cycle that it is decoded. Therefore, the MAC can have the information it needs in 9.91 μs after packet reception starts — the receiver pipeline processing latency.

If one were to transmit the entire MAC header at the payload rate, then the MAC header would incur an additional latency of 11.8 μs, due to the additive latency of the PHY header decoder. Even this larger delay is small compared to the typical packet duration and dwarfs the latency of passing information from hardware PHY to a software MAC.

4.4.2 Runtime Reconfiguration through Interrupts

In this experiment, we measure the latency of reconfiguring the PHY to abort ongoing reception and switch to transmit mode. This mechanism is useful in CMAP, where the MAC must first receive the headers of the ongoing transmission, and then switch to transmit mode if its pending transmission does not conflict with the ongoing transmission. This experiment shows that Airblue allows higher layers to interrupt and reconfigure lower layers at runtime with very small delays.

We implemented the reconfiguration in this experiment using the request-response mechanism for performing coordinated reconfigurations (Section 3.3.2). The *Interceptor* in the MAC first sends an abort request to the head of the receiver pipeline (*RX Controller* in Figure 3-7). The controller then injects a special "abort token" into the pipeline and discards the remaining received data. Every module that receives the token flushes the state of the packet it was receiving before. When the abort token reaches the end of the pipeline, the *RX Controller* sends a response to the *Interceptor* to indicate the completion of the abort. By resetting the state of the receiver pipeline, the correctness of future receptions is guaranteed. It is then safe for the MAC to initiate a new transmission.

Implementing the abort mechanism described above required changing 115 lines of code in the controller at the head of the pipeline, and did not require modifications to any other modules in the baseband PHY. The simplicity of this modification was the result of our stream control mechanism in the pipeline — the pipeline modules

expect a control token which demarcates the tail of the packet. We inject an abort token by sending a tail of packet control token.

The measured delay between the abort request and response at the MAC is equal to the time it takes for the abort token to travel along the pipeline from the *RX Controller* to the end, which is equal to 3.04 μs when using the Viterbi decoder and 4.67 μs when using the BCJR decoder. These latencies could be improved by reimplementing the decoder modules to support a faster flush. Once again, we note that such quick reconfigurations of the PHY by the MAC cannot be performed if either the MAC or the PHY is implemented in software.

4.5 Summary

Through the implementation of a variety of relatively complex protocol changes, we have demonstrated that Airblue is easy to modify and that, when modified, it meets the performance requirements of current wireless protocols. In particular, we can easily modify the platform to send per-packet feedback, implement new decoding algorithms, and perform runtime reconfigurations of the pipeline, all while meeting 802.11 timing requirements. Moreover, we show that simulations and on-air experimentations are both important to validate a protocol. As Airblue provides supports to achieve both under a unified framework, we believe that it is an ideal platform for cross-layer protocol development.

Chapter 5

Dynamic Parameterizations

Most parameters supported by the original Airblue implementations are compile-time parameters. Every change to the values of those parameters implies a recompilation and a resynthesis of the complete design. On the other hand, the values of run-time parameters can be changed by the external software written in C or C++. Generally speaking, a design with more run-time parameters means it is more flexible to implement adaptive protocols. Moreover, from a testing perspective, recompiling compile-time parameterized designs can take hours, which is much longer than the compilation time of run-time parameterized designs that usually only takes seconds. In this chapter, we discuss several techniques which allow designers to reuse existing compile-time parameterized designs to implement run-time parameterized designs.

5.1 Introduction

In Airblue FPGA implementation, there are two types of parameterizations: static parameterizations and dynamic parameterizations. Parameters of the former are configured at compile-time and their values cannot be changed at run-time. Different from software where the overhead of supporting dynamic parameters over static parameters is usually negligible, static parameters in hardware can significantly reduce the logics complexity. For example, most protocols only require a small set of permutations for its interleaver. By using static parameterization to describe each

permutation, the compiler can optimize each permutation into just wires. On the other hand, dynamic parameters provide more flexibility than static parameters by allowing the values to be updated at run-time. Because of this reason, there are many advantages to have a module to support dynamic parameters instead of static parameters: First, many cross-layer protocol proposals can only be evaluated with libraries that are dynamically parameterized. It is because these protocols will need to be able to configure the baseband processor into different modes frequently in microsecond time scale, which is difficult to be achieved by FPGA partial reconfigurations. Second, architectural study on efficient dynamically parameterized implementations can have huge impact on the designs of future commodity ASICs. Mobile devices have been supporting more wireless protocol standards in each generation and the trend is expected to continue into future generations. Conventional mobile System-On-Chips (SOCs) have separate modems for different wireless standards. For example, TI's OMAP 4 platform, whose block diagram is shown in Figure 5-1, has 4 modems for GPS, WiFi, Bluetooth and 3G/4G respectively. It will be interesting to study whether it is possible to have designs that can be shared across these different standards so that the area and power of the chip can be reduced. Finally, we can increase experimentation speed by reducing the amount of FPGA recompilations. It takes hours to generate a new FPGA image when we change the values of some static parameters in the Airblue system because those changes affect the generated RTL. On the other hand, if we implement those parameters such that they can be configured dynamically from external software, then we will be able to evaluate new configurations by changing only the software whose compilation speed is usually in seconds.

One drawback of dynamic parameterization is that a dynamically parameterized implementation sometimes can be much more complex than its statically parameterized counterpart. Take the interleaver as an example again: if one wants the interleaver to support all possible permutations dynamically while maintaining the same throughput of one permutation per cycle as the static version, one might implement a Beneš network [7] whose implementation requires $O(NlogN)$ 2x2 switches. Obviously, this implementation will have much higher area cost when compared to

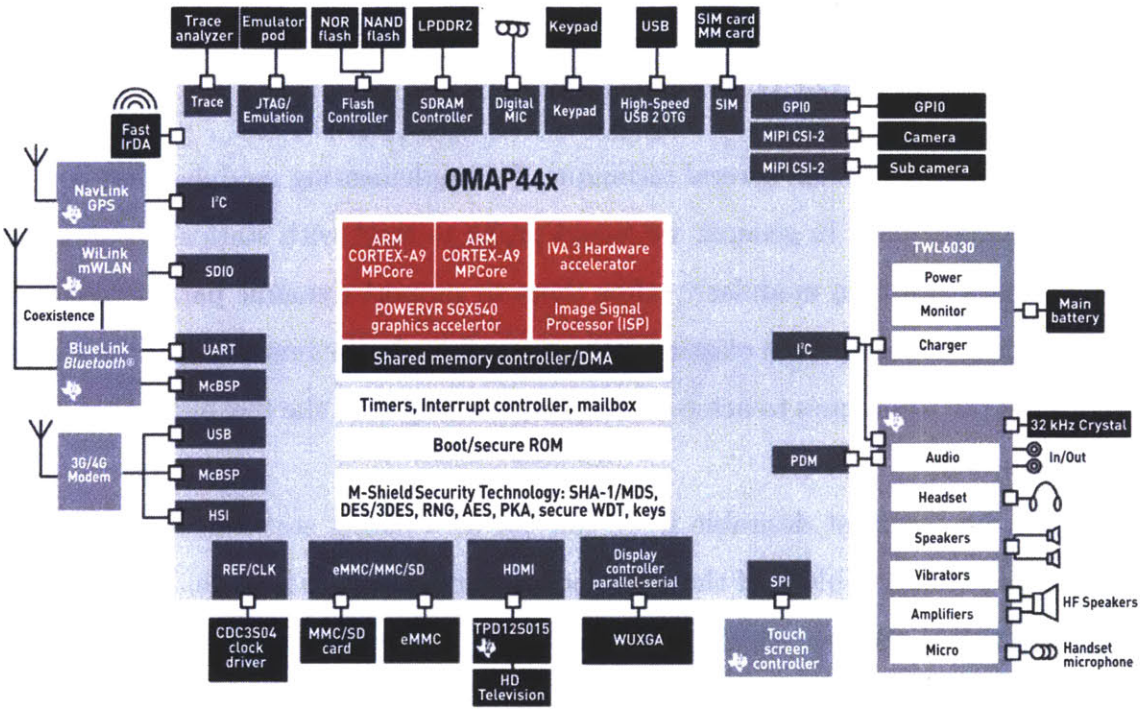


Figure 5-1: Block diagram of the TI's OMAP 4 Mobile Application Platform (Figure courtesy of TI)

the implementation with static parameterization which are mostly composed of wires.

As FPGA area was a big concern at the beginning of the project, most of the parameters in the Airblue library were implemented as static parameters. However, as there is an increasing need to provide modules with increasing number of dynamic parameters for future cross-layer protocols experiments, we plan to gradually refine the current Airblue libraries to support more dynamic parameters. There are several techniques we can use to implement dynamically parameterized modules. We are going to discuss them in details in the next section.

5.2 Techniques for Implementing Dynamically Parameterized Modules

In this section, we discuss several techniques for implementing modules that support dynamic parameters. In general, we would prefer to start with statically parameterized modules and then modularly refine them to support dynamic parameterization. The goal is to reuse as much existing code as possible. In the remaining of the section, we discuss the techniques to achieve this goal, starting with the technique that allows maximum reuse.

It will be the most desirable if we can use an existing statically parameterized module as a building block of the dynamically parameterized module. Usually, there are two possible ways to achieve this, which will be explained as follows:

5.2.1 Filling in Input And Puncturing Output

Sometimes, it is easy to use a module instantiated with a large static parameter value to generate results for smaller dynamic values by expanding the module inputs by filling in default values and puncturing the module outputs. FFT is a good example. One can use a larger point FFT to generate the results of smaller-point FFTs using this technique. Let us look at how to use a FFT of N points to produce the results of a FFT of $\frac{N}{2}$ points, i.e., given the a module implementing Equation 5.1, obtain the results of Equation 5.2 by manipulating X_k and x_n.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi k \frac{n}{N}} \tag{5.1}$$

$$X_k = \sum_{i=0}^{\frac{N}{2}-1} x_i e^{-2\pi k \frac{i}{\frac{N}{2}}} \tag{5.2}$$

It should be obvious that the exponential terms in both equations are equivalent when $n = 2i$. Therefore, if we set x_n to x_i when n is even and $n = 2i$ and fill in zeros for x_n when n is odd, then the first $\frac{N}{2}$ X_k will be results of FFT of $\frac{N}{2}$ points and

we can discard the other half. By induction, we can generate the result of any FFT with $\frac{N}{2^t}$ points for any integral t as long as $Nmodulous2^t = 0$ using a N-points FFT. The pseudo-code of the inputs filling and output puncturing algorithms are shown in Figure 5-2.

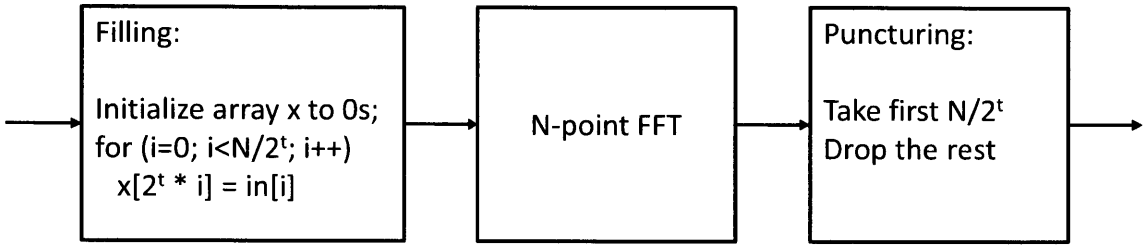


Figure 5-2: Pseudo-code of the input fillings and output puncturings for a N-point FFT to produce the result of the $\frac{N}{2^t}$-point FFT

The main advantage of the input fillings and output puncturings approach is the implementation simplicity. In general, the state machines implementing the filling and puncturing algorithms are much simpler than the state machine of the statically parameterized module itself. Another advantage is performance predictability. Usually, the latency and throughput at different parameter values are obvious with the derived design. For example, in the case of the FFT above, if the FFT kernel can produce a result every t cycles, the derive module will be able to produce the result of any smaller FFT every t cycles, thus allowing us to calculate the throughput.

The disadvantage of this approach is inefficiency. It is inefficient both in terms of energy and performance because our dynamic FFT implementation performs all the computations required by the FFT kernel to get the results of smaller FFTs.

Many modules in the Airblue can use this technique to support dynamic parameters. Some examples are the Channel Estimator, the Mapper and the Demapper.

5.2.2 Using Statically Parameterized Module as Micro-Kernel

Apart from supporting smaller dynamic parameter values using a module instantiated with a large static parameter value, it is also possible to achieve the opposite. Let us consider the FFT example again. It is well known that any FFT can be decomposed into smaller FFTs. A very popular decomposition is the Cooley-Tukey algorithm [19]. A specialized case of Cooley-Tukey is the radix-2 decimation-in-time (DIT) FFT, which states that a N-point FFT can be computed by first computing the results of two $\frac{N}{2}$-point FFTs, followed by multiplying the results of the second FFT by twiddle factors, and finally merging the results by computing $\frac{N}{2}$ 2-point FFTs. Figure 5-3 shows the data flow diagram of the radix-2 DIT FFT when $N = 8$.

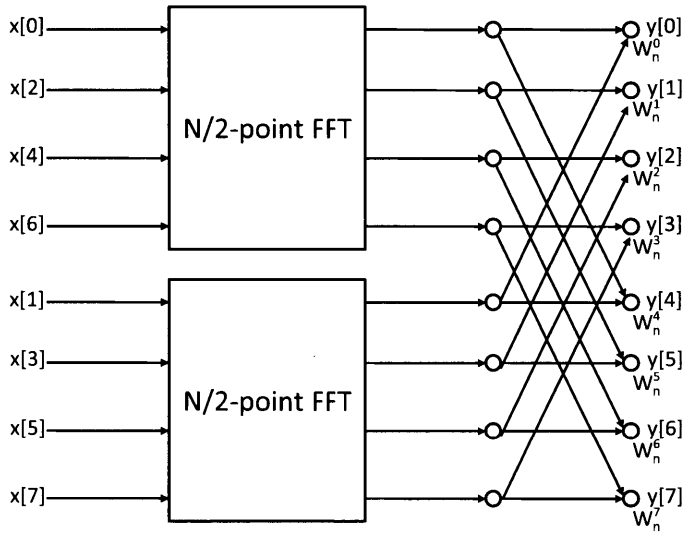


Figure 5-3: Data flow diagram of the radix-2 DIT FFT when $N = 8$

Assuming N is power of 2, if we recursively apply the radix-2 DIT FFT algorithm on the $\frac{N}{2}$-point FFTs, we will get the well-known radix-2 FFT implementation. In this case, we only need to instantiate multiple 2-point FFTs, twiddle factor generators, twiddle factor multipliers and memories to save the temporary result to implement a dynamically parameterized FFT.

When compared to the filling and puncturing approach, the micro-kernel approach usually uses the statically parameterized module more efficiently with the tradeoff of

more complex state machines.

5.2.3 Reusing Submodules of Statically Parameterized Modules

Sometimes, it is hard to reuse the whole module as inner kernel. However, it may still possible to reuse some internal submodules. For example, assume we would like to modify the Viterbi decoder presented in Chapter 2 to support multiple constraint lengths, then we can reuse the implementation of the Add-Compare-Select (ACS) unit presented in Figure 2-14 on page 46 to implement the Path Metric Unit (PMU). As shown in Figure 2-13 on page 45, PMU for constraint length k requires 2^{k-1} ACS computations and a permutation for 2^k elements where the content of the input vector at index x is copied to the content of the output vector at index y where y is equivalent to circularly right shift the value of x by one, i.e., right shifting x by one and also copy the original value of the least significant bit to the most significant bit. Figure 5-4 shows an implementation of the PMU that can support multiple constraint lengths. In the design, there is a memory that store the 2^k path metrics. The maximum k supported depends on the size of the memory. At the heart of the PMU, it is the computation unit which contains bunch of ACSs. There are two other modules around the computation unit. One reads chunk of memory sequentially to provide the path metrics required by the computation unit. Another takes the result from the computation unit and writes them back to the memory according to the permutation. Also, the computation unit also takes the branch metrics provided by the external Branch Metric Unit (BMU) to produce the result. Note that in this design, the state machines of the two modules that access the memory needed to be changed dynamically according to the provided constraint length k but the behavior of the computation unit is independent of k.

It should be apparent that there are abundant opportunities to reuse submodules in statically parameterized designs when we are implementing dynamically parameterized version of the modules because they usually involve the same algorithms.

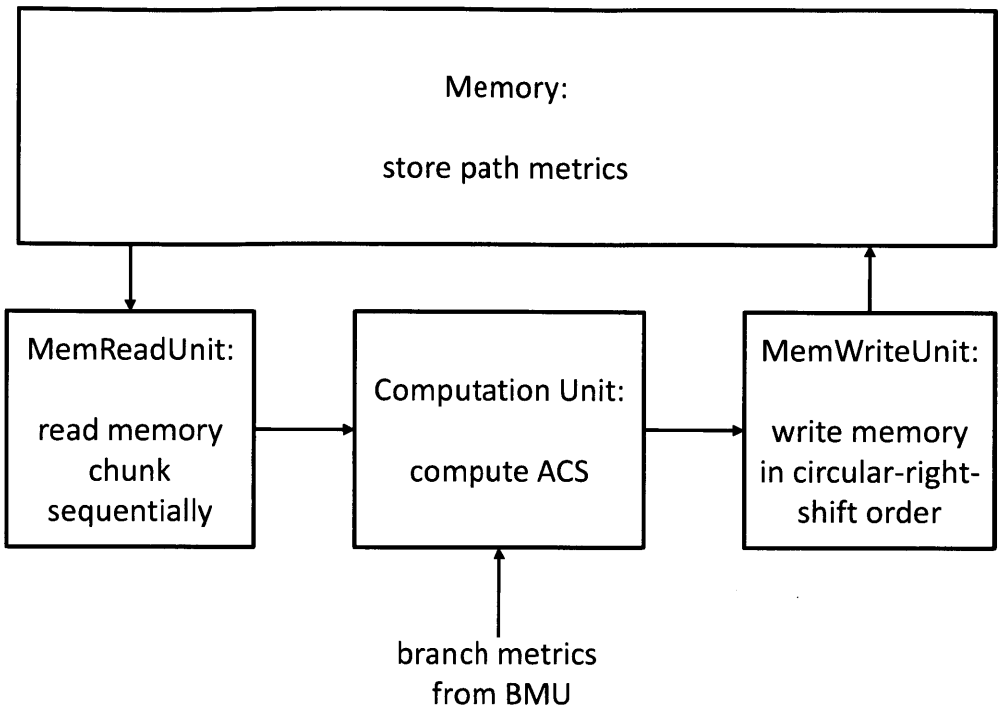


Figure 5-4: Implementation of a Path Metric Unit that supports dynamic reconfiguration of constraint length

Examples include but are not restricted to: auto-correlators, CORDICs, and cross-correlators in Synchronizer; the inverse-square-root calculator in Channel Estimator; radix-2 butterflies in FFT; Linear Shift Registers (LFSR) in Scrambler.

5.2.4 Using New Algorithms or Architectures

Occasionally, a design that makes sense for a statically parameterized implementation cannot be translated into an efficient dynamically parameterized implementation. Permutations are good example. Let us consider the circular right-shift permutation again. Figure 5-5 shows a statically parameterized implementation of this permutation. As we can see, the implementation contains a deserializer which gather elements. Once enough elements are collected, the whole group will be sent to the permutation network which consists of only wires. Then a serializer takes the output from the network and serializes the data for the output stream. While it is easy to make the

serializer and deserializer to dynamically support multiple permutation sizes, it is more difficult to transform the hardwired permutation network accordingly. If the number of permutations needed is small, one solution is to instantiate multiple hard-wire permutations and choose the right one to use dynamically using mux and demux. On the other hand, if we need to support many permutation sizes, a more efficient architecture is needed. First, we need to understand the semantics of the circular right shift permutation with 2^k elements where k is a natural number. We notice that the permutation is effectively moving all the even-number-indexed elements to the first half and the odd-number-indexed elements to the second half. As a result, the architecture presented in Figure 5-6 will be able to perform circular right shift permutation effectively. In this architecture, there are two fifos. The inputs are en-queued alternately between the fifos. The outputs are generated by reading the first fifo consecutively for the half of the permutation size followed by reading the second half from the second fifo. The permutation size can be changed easier by controlling the number of elements to be read from the fifos consecutively before switching. Note that our implementation will work not only for permutation of size 2^k but all the permutation sizes which are even.

In Airblue, there are many kinds of permutations involve in various stages of the pipeline such as FFT/IFFT, interleaver/deinterleaver, Viterbi. Efficient implementations for these permutations supporting multiple sizes require some innovations in architecture or algorithms.

5.3 New Scheme for Data-Driven Control

In the last section, we discuss several techniques of implementing dynamically parameterized modules. With an increasing amount of dynamically configurable parameters in the system, the old way of implementing data-driven control is no longer adequate. In this section, we discuss a new architecture for data-driven control. In this architecture, we introduce a generic control forwarding structure which wraps around all the dynamically parameterized modules in the system. The goal of this structure is

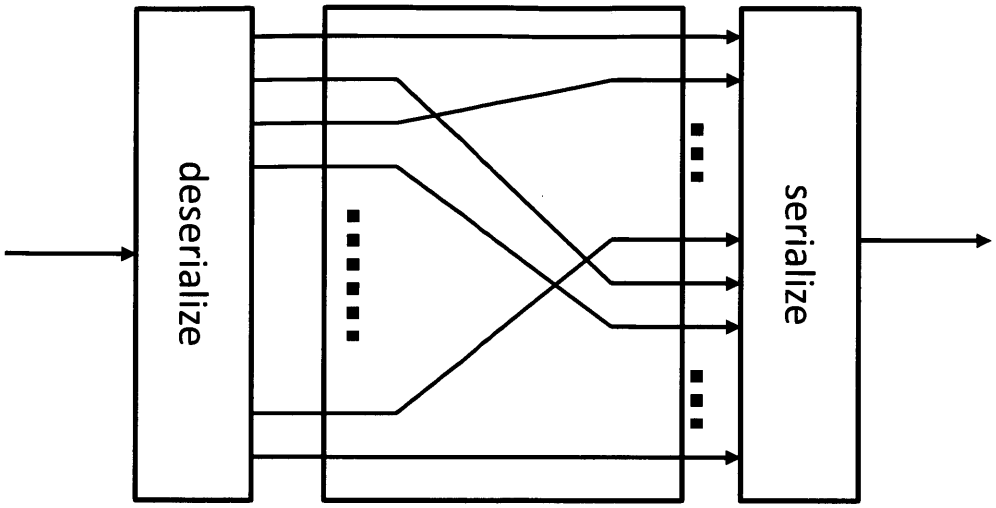


Figure 5-5: Implementation of a fixed size circular shift permutation

to free individual modules from knowing parameters that are not intended for them.

In Chapter 3, we demonstrated why our latency-insensitive transceiver pipelines require data-driven control where control information are embedded into the datapath along with the data. Similarly, the same technique is also used for passing the lower-to-higher-layer information.

There are two obvious ways of implementing data-driven control which are shown in Figure 5-7. The first scheme is the tightly-coupled scheme where a datum is always associated with its corresponding control as a bundle. When a module in the pipeline receives a bundle, it processes the data according to the control information it extracts from the bundle. After the data processing, it creates a new bundle which contains the same control information with the processed data and possibly some internal state of the module representing the lower-to-higher-layer information. Then, it sends the new bundle down the pipeline.

The second scheme is the loosely-coupled scheme. In this scheme, control and data are represented as different types of tokens. Reconfiguration is achieved by first sending the control tokens down the pipeline followed by the data tokens that should be processed under the new configuration.

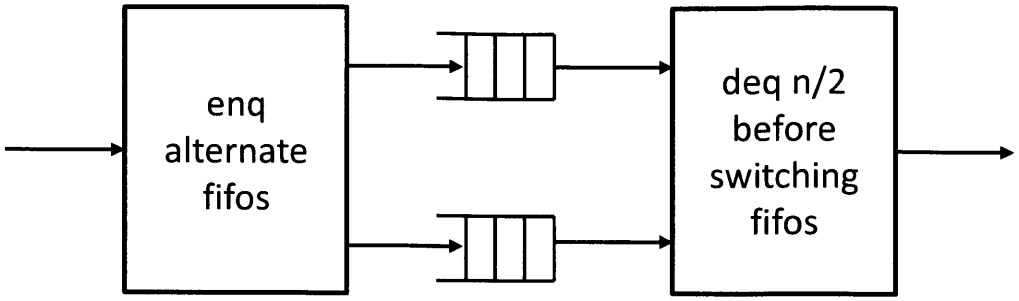
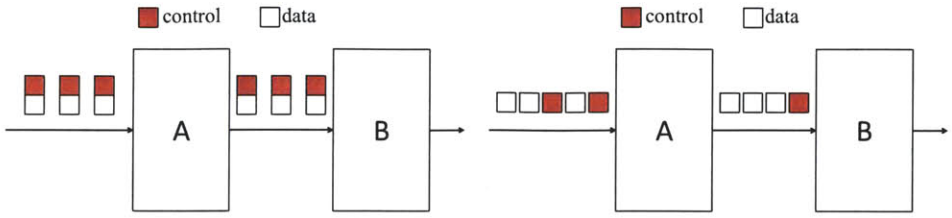


Figure 5-6: Implementation of a Path Metric Unit that supports dynamic reconfiguration of constraint length

5.3.1 Limitations of Current Architecture

We chose to implement the tightly-coupled scheme in the current Airblue architecture. It is because bundles in the tightly-coupled scheme can be easily represented as a "c-like struct" in Bluespec. Moreover, no extra cycle is wasted in the interconnect for passing the control information. However, we noticed two major limitations of the tightly-coupled scheme:

- **Large Bundle Size:** In the current architecture, it always sends the whole bundle down the pipeline. Additional control or other cross-layer information is supported by adding fields to the struct. The main advantage of this approach is that it simplifies the effort on performance guarantee. This is because we simply widen the dataline to accommodate the extra information. Therefore, if the original design meets the performance target, the new design will be able to do so too. However, this approach becomes unrealistic when the protocol being implemented needs to pass a lot of cross-layer information which makes the bundle too large. In this scenario, the better approach would be splitting a bundle into multiple fixed-size sub-bundles for communication between modules. Note that one way of splitting the bundle is to have different types of sub-bundles for control and data respectively, which essentially implements the loosely-coupled scheme.



(a) Tighly-coupled: each datum is always associated with the corresponding control. When a module receives a bundle, it will extract the data and process it according to the control in the bundle

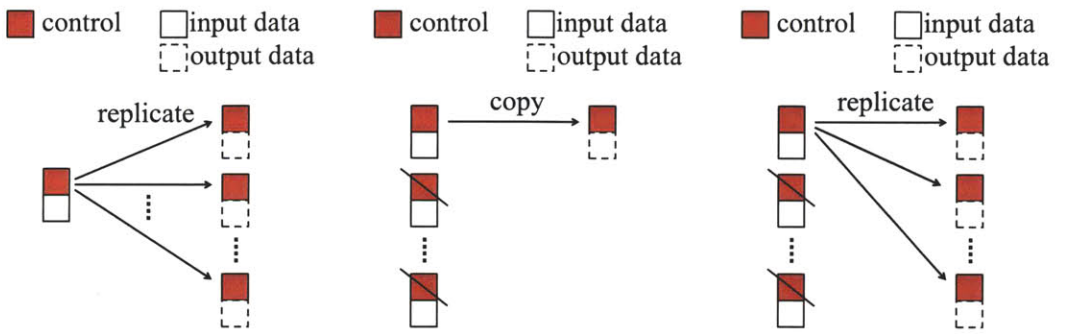
(b) Loosely-coupled: there are two types of tokens: control tokens and data tokens. An operation in new mode normally starts by sending control tokens down the pipeline to configure the modules before sending data tokens that should be processed in this mode

Figure 5-7: Two different schemes for data-driven control

- **Difficult Control Forwarding:** There is no ambiguity of how to pass the control correctly under the tightly-coupled scheme if the following assumption about the pipeline is true: For each module in the pipeline, it consumes one input bundle to produces one output bundle which in return corresponds exactly to the input bundle required for the next module. Under this assumption, each module just needs to forward the control of the input bundle to the corresponding output bundle. On the other hand, how should the module forward the control information if such assumption is invalid, e.g. the module consumes three bundles to produce two bundles? In this scenario, the correct forwarding strategy depends on understanding how the subsequent modules use the control information, thus making the design not completely modular. To make the problem even more challenging, some modules in the pipeline may change the input-to-output ratio at different operation modes. A good example is the puncturer in 802.11 a/g which removes different fractions of bits from the input stream under different transmission rates. Fortunately, for our implementation of 802.11 a/g, we were able to make most modules conform to the assumption by passing data at the granularity of an OFDM symbol. For the several modules that change rates, we adopted the following control forwarding strategies:

If a module generates multiple bundles from an input bundle, it replicates the input control to each output bundle as shown in Figure 5-8(a); Vice versa, if a module requires multiple input bundles to generate an output bundle, the control of the output bundle is simply copied from the control of the first input bundle as shown in Figure 5-8(b). In the case where a module consumes multiple input bundles to generate multiple output bundles, we replicate the control of the first input bundle to all its output bundles and discard the rest as shown in Figure 5-8(c). Note that the correctness of the last two schemes rely on the assumption that the controls discarded do not contain additional information because they are the same as the first control in the bundle. For reconfiguration purpose alone, we achieve this by carefully setting the input and output width of these modules and ensuring that no reconfiguration is performed within the symbol boundary, which is adequate for most foreseeable protocols (most current protocols only reconfigure between packets). However, it is much harder to maintain the assumption for passing cross-layer information purpose because modules may want to pass the information at finer granularity. For example, let us assume the channel estimator operates at subcarrier granularity (where data from multiple subcarriers together form a symbol) and associates each of its output with the subcarrier characteristic as the cross-layer information. On the other hand, the subsequent demapper operates at symbol granularity, thus gathering multiple bundles before generating the output. In this scenario, our forwarding scheme will incorrectly remove useful information. Our solution to this problem is to force each module to only send cross-layer information at the beginning of a symbol boundary like the control. Again, this approach makes the design less modular. Moreover, it exacerbates the large bundle size problem because each bundle will now need to be able to fit a whole symbol worth of cross-layer information.

Although the problems created by the limitations of the current architecture have been manageable so far, we expect the problems to exacerbate when implementing protocols that pass increasing amount of cross-layer information. Therefore, we are



(a) Splitting controls: replicate the control from input bundle to all output bundles

(b) Merging controls: copy the control of the first bundle to the output bundle

(c) Split-And-Merge controls: replicate the control of the first bundle to all output bundles

Figure 5-8: Three types of control forwardings for the tightly-coupled scheme: Splitting and Merging

presenting a new architecture that utilizes loosely-coupled control and a more generic control forwarding scheme to tackle the limitations.

5.3.2 Proposed Architecture

In this section, we are presenting a new architecture that utilizes loosely-coupled control and a more generic control forwarding scheme to tackle the aforementioned limitations. First, we start with explaining the new message type in Section 5.3.2 followed by the discussion of the new module interface in Section 5.3.2. We finish by presenting the architecture of the generic control forwarding logics.

Message Type

As mentioned previously, we implement the loosely-coupled control in the new architecture. We identify three types of messages: `Control` messages, `Statistic` messages and `Data` messages. `Control` messages are used to configure a module or multiple modules in the transmit or receive pipelines; `Statistic` messages are used to represent data that are generated by a module and will be passed down the pipeline being intact until they reach the controller, they are useful for delivering cross-layer

information; On the contrary, `Data` messages are used to represent data that are generated by a module and will be processed by the next immediate module in the pipeline. The new approach requires us to modify the type definition of Mesg shown previously in Figure 2-7 to Figure 5-9. The new type definition is implemented using `tagged union` instead of `struct`. `Tagged union` is useful to distinguish different types of data by giving each type a unique tag. In our sample whose definition is shown in Figure 5-9, we use three tags (`Ctrl`, `Stat` and `Data`) to separate the aforementioned three types of messages. Moreover, the types of data they contain are of types `CtrlMesg`, `StatMesg` and `dataMesg` respectively. Note that `dataMesg` is a type variable to reflects the fact that data are usually transformed into different representations by DSP algorithms along the pipeline. In the remaining of 5.3.2, we explain in details about the `CtrlMesg` type and the `StatMesg` type.

```
typedef union tagged {
  CtrlMesg Ctrl;
  StatMesg Stat;
  dataMesg Data;
} Mesg#(type dataMesg);
```

Figure 5-9: New type definition of Mesg

CtrlMesg

In our proposal, each module in the design is assigned a unique id. Moreover, each module can contain one or more control registers that can be reconfigured using control messages. Figure 5-10 shows the type definition of `CtrlMesg`. There are five types of control messages: `ReadCmd`; `WriteCmd`; `StartPacket`; `EndPacket`; `AbortPacket`. `ReadCmd` reads the control register `reg_id` of the module with id `dst_id`. The result will be passed as a `StatMest`. `WriteCmd` writes the control register `reg_id` of the module with id `dst_id` with the value `operant`. `StartPacket` indicates the start of a packet. It is a broadcast message that informs all the module that they should be configured into the specific mode defined by the value of the `operand`. When a module receive this message, it will read its instruction memory dedicated for the re-

quested mode to configure all the necessary control registers according to the `operand` and then forward the control message to the next module in the pipeline. Note that the content of the instruction memory itself could be modified using the `WriteCmds`. `EndPacket` indicates the end of a packet. Similar to `StartPacket`, it is a broadcast message. When a module receives this message, it forwards the current result to the next module and then switches to idle. `AbortPacket` indicates an abortion. It behaves similarly to `EndPacket` except that the module can receive this command at an unexpected state and therefore, the module should consult its instruction memory dedicated for abortion to see how to handle the abortion.

```
typedef union tagged {
  struct {
    Bit#(8) dst_id; Bit#(16) reg_id;
  } ReadCmd;
  struct {
    Bit#(8) dst_id; Bit#(16) reg_id; Bit#(32) operand;
  } WriteCmd;
  struct {
    Bit#(32) operand;
  } StartPacket;
  void EndPacket;
  void AbortPacket;
} CtrlMesg;
```

Figure 5-10: Type definition of CtrlMesg

StatMesg

Figure 5-11 shows the definition of StatMesg. It is defined as a `struct`

```
typedef struct {
  Bit#(8)  src_id;
  Bit#(16) reg_id;
  Bit#(32) reg_val;
} StatMesg;
```

Figure 5-11: Type definition of StatMesg

Module Interface

As the original module interface definition presented in Chapter 2 is already polymorphic to the types of its input and output messages, one may expect the module interface does need to be modified because we can simply set the type parameter to use the new message definition. This is mostly true except that we add two output methods to the module interface, namely `num_in_need` and `num_out_rdy`. The former indicates the number of inputs required to generate the next batches of outputs while the latter indicates the number of outputs that are going to be ready to be consumed. These two methods are added to provide information that can help forwarding the control information. More details will be explained in Section 5.3.2 when we discuss the implementation of a module and its control forwarding logic.

```
interface Block#(type in_t, type out_t, numeric type in_sz, numeric type
    out_sz);
  interface Put#(Mesg#(in_t))  in;
  interface Get#(Mesg#(out_t)) out;
  method    Bit#(in_sz)        num_in_need;
  method    Bit#(out_sz)       num_out_rdy;
endinterface
```

Figure 5-12: New type definition of Block

Implementations of the Module and the Control Forwarding Logics

As mentioned above, the module interface now contains two more methods. Therefore, the module implementation should also be modified in order to expose the required information. The `num_in_need` returns the additional number of input data messages required before it can produce the next bundle of output messages. For example, if the module is able to produce an output bundle for each input message, then `num_in_need` will always return zero. In another example, if the module produces an output bundle for every two input messages, then `num_in_need` will be initialized to one and count down circularly whenever an input data message is received. It should be obvious that the idea can be generalized to any input bundle size by mak-

ing `num_in_need` a count down variable with initial value set to the input bundle size minus one. Also note that a module may have different initial values at different modes.

The `num_out_rdy` returns the number of output messages that are ready to be consumed by the next module. Again, we can implement this with a counter. The counter is decrement by whenever an output is consumed and is incremented by the output bundle size plus the number of statistic messages whenever the module start processing a new input bundle, i.e.reading an input while `num_in_need` is zero.

Control-Forwarding Logics

Figure 5-13 shows the architecture of the control-forwarding logics. We can see in the figure that the control-forwarding logics is implemented as a wrapper around an Airblue module. Before the module, there is a dispatcher which dispatches the incoming messages either to the module or to a reorder buffer. After the module, there is a merger which merges the messages from the reorder buffer back to the output messages of the module. When the dispatcher receives an incoming message, it first checks what the message type is. If the message is a control message that is not intended for the current module, i.e., a `ReadCmd` or a `WriteCmd` whose `dst_id` is different from the current module, or a statistic message, the message will be transferred to a reorder buffer. For all other messages, they will be passed to the module through the `in` interface. The idea of having a reorder buffer is to move the control or statistic messages to the correct output bundle boundary: If a module receives one of these messages while the module is processing an output bundle, the merger will delay the insertion of the received message to the end of the output bundle after it is produced. In our implementation, the merger achieves this by taking messages from the reorder buffer when both methods `num_in_need` and `num_out_rdy` return zeros. Otherwise, the merger takes message from the output of the module until both methods return zeros. Also, to make sure this scenario will eventually happen, the dispatcher stops putting messages into the module if the reorder buffer is not empty and the `num_in_need` method returns zero. This ensures that the module will not start processing a new input bundle until all the messages in the reorder

buffer are forwarded following the end of the previous bundle.

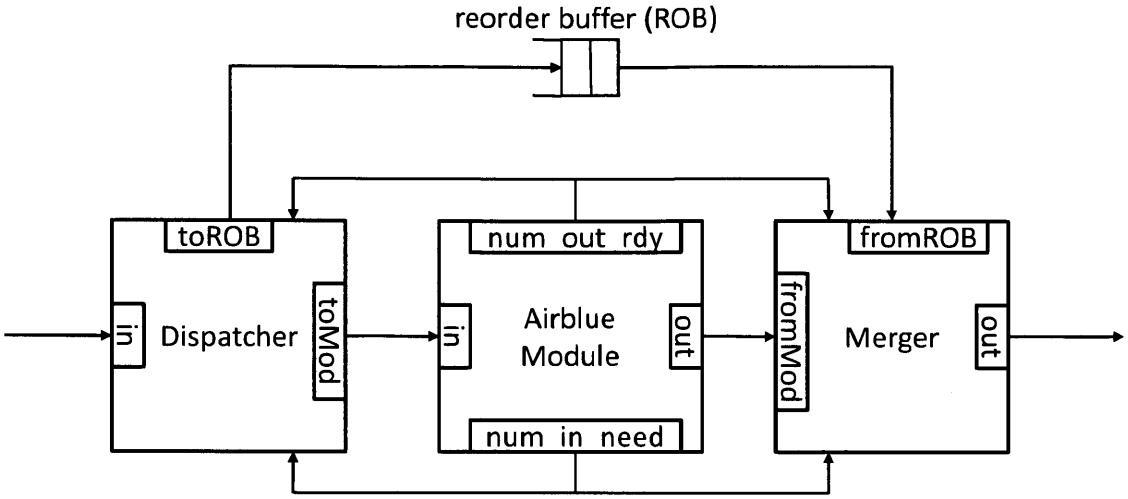


Figure 5-13: The architecture of the control-forwarding logics

Concurrency and Deadline Issues

For our forwarding scheme to make forward progress, it requires that the module to be able to generate the next bundle of output messages by receiving enough input data messages. While doing so, all the control messages or statistic messages received in between will be moved to the reorder buffer. However, if there are too many of such messages causing the reorder buffer to be full, our approach will encounter deadlock caused by head-of-line blocking. As a result, the size of the reorder buffer should be adjusted appropriately to make sure this will not happen. Moreover, even the reorder buffer is never full, significant performance loss due to concurrent issues can still occur when there are too many control or statistic messages not being passed at the bundle boundary. Remember that the dispatcher stops dispatching data message to the module once the reorder buffer is not empty and the `num_in_need` method returns zero. After that, it only resumes dispatching when the reorder buffer is empty again. For modules that are highly pipelined but with long startup latencies, our proposed forwarding scheme will cause the module to suffer from an addition startup latency every-time a message gets into the reorder buffer. The solution is to send control

messages less often and statistic messages in a bursty manner. If this is not possible, a more complicated control-forwarding scheme may be necessary.

5.4 Discussion

In this chapter, we discuss several techniques for implementing dynamically parameterized modules as well as a new control forwarding architecture. Although dynamic parameters provide more flexible than static parameters, the latter usually achieve better performance with lower hardware cost. As a result, deciding whether a parameter should be implemented dynamically or statically is a tradeoff between flexibility and complexity. When a developer converts a static parameter into a dynamic parameter, there are two main questions to consider: First, what is the expected range of values supported by the dynamic parameter? As mentioned, flexibility comes with the cost of hardware complexity. Therefore, the developer should pick a range that can cover expected future protocols with a reasonable estimated implementation cost. Once the range is picked, the second question is what is the right design for the dynamic parameter? Does any of the implementation technique discussed in this chapter applicable? If not, is it possible to come up with innovative efficient architecture for the parameter? Sometimes, the developer may find it hard to come up with an efficient architecture for a dynamic parameter expected to support a wide range of values, e.g., an interleaver block that supports all permutations with variable sizes. In this case, he or she may need to narrow the range and repeat the process.

Chapter 6

Comparison of High-Level Design Methodologies for Algorithmic IPs

Airblue was implemented using a relatively new high-level hardware description language called Bluespec. This language decision may not appeal to protocols designers who usually model the digital signal processing (DSP) algorithms in software languages like MATLAB or C/C++. Since these algorithms are usually highly structured and parallel, there exist tools trying to generate efficient hardware designs from automatic synthesis of algorithms written in such languages. In another approach, Bluespec allows hardware designers to express the intended micro-architecture through high-level constructs. In this chapter, we compare a C-based synthesis tools against Bluespec in developing hardware for Reed-Solomon decoding algorithm under area and performance metrics. Although we found that C-based synthesis tools design flow can be effective in early stages of the design development, the Bluespec design flow produces better final hardware. This is because in later stages, designers need to have close control over the hardware structure generated which is difficult to express under the constraints of sequential C semantics.

The goal of this chapter is to show that current C-based tools do not provide designers enough expressive power to express important design properties (e.g. the structuring of state elements, the pipelining of inter-module communication, and the fine-grained resource arbitration). Therefore, they are not suitable for developing

high-quality hardware designs for the Airblue platform. On the other hand, we also show that these properties can be expressed naturally in high-level HDLs like Bluespec, while maintaining the inherent algorithmic modularity.

The content of this chapter is based on a journal article published in the 2010 IEEE Embedded Systems Letters (Volume: 2, Issue: 3). Abhinav Agarwal and I are the co-authors of this article. This research is funded by Nokia Inc. (2006 Grant to CSAIL-MIT)

6.1 Introduction

Wireless communication systems usually involve a wide range of DSP algorithms where each requires huge amounts of computations at reasonable power budgets. To satisfy these requirements, Airblue implements these DSP algorithms in dedicated hardware on FPGA. Meanwhile, DSP algorithms are often modeled using MATLAB or C/C++ because of the familiarity of algorithm designers with these languages and because it frees the designers from hardware considerations. A tool that could generate efficient hardware directly from these high-level codes would be ideal for implementers. Such a tool, however, must provide its users with mechanisms to control generated hardware so that implementations with different performance, area and power tradeoffs can be generated according to the usage of the DSP algorithm. Several EDA vendors provide tools for this purpose [14, 58, 83, 82].

C-based tools fall into two distinct categories - those that adhere to pure C/C++ semantics like Catapult-C [58], PICO [82] and C-to-Silicon Compiler [13], and those that deviate from the pure sequential semantics by allowing new constructs, like SpecC [28], SystemC [66] and BachC [48], (see [25] for a detailed discussion of this topic). The tools from the first categories are the most appealing because they require the least amount of source code modifications to generate hardware when given a software implementation.

There are reasons to be optimistic about such C-based tools: DSP algorithms are usually highly structured and have abundant fine-grained parallelism. This is exactly

the kind of parallelism where extensive research in parallel compilers over the last three or four decades has shown the greatest success. Since hardware generation is mostly about exploiting fine-grained parallelism, the rich body of compiler research should be directly applicable to the task at hand. The practical question is *can the compiler or the tool be directed to generate hardware that satisfies some specific area, performance or power metric.* The economic importance of this question is obvious: if C-based synthesis cannot lead to the ultimate target design then at some stage, the implementers will have to jump the rail at enormous design cost and resort to RTL design and synthesis tools.

Most C-based synthesis tools provide mechanisms to control the generated hardware. General constraints, such as the operating frequency, can be applied to the whole design. These tools automatically explore the design space to find a design that satisfies the constraints. The user can give annotations, such as loop unrolling, regarding how to optimize a particular part of the program without affecting the correctness of the code. Hardware designers perceive several advantages in using such a C-based design methodology [81, 34] - having a concise source code allows faster design and simulation, technology-dependent physical design is relatively isolated from the source and using an untimed design description allows high level exploration by raising the level of abstraction.

Most implementers of DSP algorithms regard designing at the RTL level (as in Verilog or VHDL) as tedious, inflexible, error-prone and slow. The high costs of ASIC and FPGA designs give credence to this belief. Bluespec [11] offers an alternative: It is a synthesis tool to help the designer express both the structure and the behavior of the desired micro-architecture at a high level. Bluespec SystemVerilog (BSV), the input language for the tool, is built on sound ideas from modern functional and object-oriented programming languages, and uses guarded atomic actions to express concurrent behaviors succinctly [16, 38]. Bluespec facilitates latency insensitive designs by automatically generating handshaking signals between components. This allows designers to incrementally refine modules to meet their requirements. We have already shown to this approach to be very efficient time and time again in previous

chapters. *However, the Bluespec tool, unlike many C-based synthesis tools, does not do any design exploration on its own - it only facilitates the expression of different design alternatives.*

In this chapter, we compare the two design methodologies via the implementation of a Reed-Solomon decoder. This example was chosen because it represented a non-trivial algorithmic IP that we wanted to add to the Airblue library as one way of performing Forward Error Correction (FEC). We were not familiar with Reed-Solomon codes before we started this project. We expected that it would be straightforward to express the desired decoder for the target performance in Bluespec. Indeed that turned out to be the case. However, even to understand the Reed-Solomon decoding algorithm, we ended up writing it in C++ first. We used this code as a golden model for the verification of the Bluespec code. We were also curious to evaluate the hardware generated from our C++ code by C-based synthesis and for this purpose we picked a popular C-based tool. With straightforward modifications to the C++ source code, and the tool's provided annotations, we were successful in quickly generating hardware but it achieved only 7% of the target throughput. Even with considerable effort we were only able to improve the design throughput to 64% of the target. The initial Bluespec design could achieve 17% of the target throughput and with modular refinements, the design's throughput improved to 504% of the target throughput, while it still used only 45% equivalent FPGA gate count as compared to the final C-based design. As a point of comparison, we compared our designs with a reference Xilinx IP core and found that the Bluespec design achieves 178% of the Xilinx IP's data rate with 90% of the equivalent gate count.

The remaining of the chapter is organized as follows: We first describe Reed-Solomon decoding algorithm in Section 6.2. The hardware implementation steps are described in Section 6.3, with a comparison of both methodologies. We discuss the Bluespec design flow through modular refinements in Section 6.4. We have discussed the language-related complexities faced during the later stages of the development in Section 6.5. Finally, the synthesis results are shown in Section 6.6 and we conclude the chapter in Section 6.7.

6.2 The Application: Reed-Solomon Decoder

Reed-Solomon codes [94] are a class of error correction codes frequently used in wireless protocols like 802.16 [44]. In this chapter we designed a Reed-Solomon decoder for use in an 802.16 protocol receiver. The decoder accepts a primitive polynomial in Galois Field 2^8, as a static variable and the number of information and parity symbols as dynamic variables. To simplify the design, our current design only supports shortened and full-length codes, and not punctured codes.

We chose the minimum throughput requirement of the design to be 134.4 Mbps, which is the maximum data rate supported by the 802.16 protocol. With the chosen decoding algorithm, 81K arithmetic operations (Bitwise XORs) are needed to process a 255 byte input data block. We used the Xilinx Reed-Solomon decoder IP version 5.1 as a baseline reference, which operates at a frequency of 145.3 MHz and can accept a new 255 byte input data block every 660 clock cycles for a data throughput of 392.8 Mbps. The target operating frequency for our Bluespec and C-based designs was kept at 100 MHz. To achieve the 802.16 protocol requirement at this frequency, the designs need to accept a new input block every 1520 cycles. To match the Xilinx IP's throughput at 100 MHz, the throughput requirement becomes 520 cycles per input block. There is some advantage in overshooting the minimum performance requirement because the "extra performance" can be used to lower voltage or operating frequency for low power implementations. However, in this study we have not pursued the low power issues beyond aiming for high performance. During the design process, our goal was to reduce number of cycles taken to process a block by increasing parallelism and to improve throughput by pipelining.

6.2.1 Decoding Process and its Complexity

Reed-Solomon decoding algorithm [60] consists of 5 main steps shown in Figure 6-1.

1. Syndrome computation

2. Error locator polynomial and error evaluator polynomial computation using

Berlekamp-Massey algorithm [8, 56]

3. Error Location computation using Chien search [18]

4. Error Magnitude computation using Forney's algorithm [27]

5. Error correction

Each input block is decoded independently of other blocks.

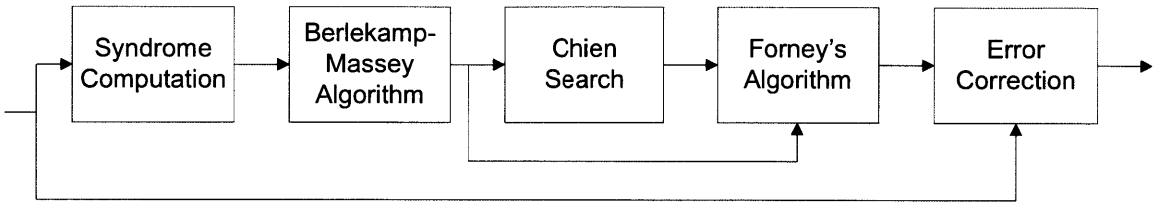


Figure 6-1: Stages in Reed-Solomon decoding

A Reed-Solomon encoded data block consists of k information symbols and $2t$ parity symbols for a total of $n(= k+2t)$ symbols. The decoding process is able to correct a maximum of t errors. Figure 6-2 shows the order of associated Galois Field (GF) arithmetic operations — addition, multiplication by a constant and multiplication by a variable.

Step	**GF Add**	**Const GF Mult**	**Var GF Mult**
Syndrome	$O(tn)$	$O(tn)$	0
Berlekamp	$O(t^2)$	0	$O(t^2)$
Chien	$O(tn)$	$O(tn)$	0
Forney	$O(t^2)$	$O(t^2)$	$O(t)$
Correction	$O(t)$	0	0

Figure 6-2: Order of GF Arithmetic Operations

Since this is a streaming application, the inter module communication has a significant impact on performance. An additional complexity arises due to difference in the Input-Output data block sizes for each step. For an input block of size n symbols, with $2t$ parity symbols and ν errors, Figure 6-3 shows the number of input and output symbols for each block. The values in Figures 6-2 and 6-3 have been derived from the algorithm described in the next section. Readers familiar with the algorithm can skip ahead to Section 6.3.

Step	Input Symbols	Output Symbols
Syndrome	n	$2t$
Berlekamp	$2t$	2ν
Chien	ν	ν
Forney	3ν	ν
Correction	$2(n-2t)$	$(n-2t)$

Figure 6-3: Input and output symbols per step

6.2.2 Algorithms and Pseudocode

An important step in the design is the translation of the mathematical algorithm into high level pseudocode. This translation is commonly done into C/C++ or MATLAB code. In the following discussion of the Reed-Solomon decoding algorithm, the input data block is denoted as

$$R(x) = r_{n-1}x^{n-1} + r_{n-2}x^{n-2} + \ldots + r_0 \tag{6.1}$$

where the parity symbols are $r_{2t-1}, r_{2t-2}, \ldots, r_0$.

Galois Field Arithmetic

Galois Fields are finite fields described by a unique primitive polynomial *pp*, with root α. For Reed-Solomon codes defined over GF 2^8, each data symbol can be described by a single byte mapped to a finite field element. Every data block consists of upto $2^8 - 1$ data symbols. For GF 2^8, arithmetic operations are defined as follows:

GF Add: A bitwise XOR operation of the data symbols.

GF Mult: This is multiplication modulo operation over *pp*. Using an iterative process, multiplying by a variable takes 15 bitwise XOR operations. Multiplication by a constant only takes at most 8 XOR operations, depending on the constant.

GF Divide: Division is a complex operation. It is commonly performed by multiplying the dividend and the inverse of the divisor, found via a lookup table.

Syndrome Computation

In this step the received polynomial, comprising of n data symbols, is used to compute $2t$ symbols known as the Syndrome polynomial using the following expression:

$$S_j = r_{n-1}\alpha^{(n-1)j} + r_{n-2}\alpha^{(n-2)j} + \ldots + r_0 \quad \forall j \in 1..2t \tag{6.2}$$

Pseudocode:

`Input:` $r_0, r_1, \ldots, r_{n-1}$

`Output:` $S_1, S_2, \ldots, S_{2t}$

`Initialize:` $S_j = 0, \ \forall j \in 1..2t$

`for` $i = n-1$ `to` 0

  `for` $j = 1$ `to` $2t$

    $S_j = r_i + S_j \times \alpha^j$

Berlekamp-Massey Algorithm

Berlekamp-Massey Algorithm computes an error locator polynomial $\Lambda(x)$ of degree ν and an error evaluator polynomial $\Omega(x)$ of degree ν for a received input vector with ν errors. $\Lambda(x)$ and $\Omega(x)$ are computed from $S(x)$ using the following relations:

$$S_j = \sum_{i=1}^{\nu} \Lambda_i S_{j-i} \quad \forall j \in \nu+1, \ldots, 2\nu \tag{6.3}$$

$$\Omega(x) = S(x) \times \Lambda(x) (mod x^{2t}) \tag{6.4}$$

Pseudocode:

`Input:` $S_1, S_2, \ldots, S_{2t}$

`Output:` $\Lambda_1, \Lambda_2, \ldots, \Lambda_\nu$; $\Omega_1, \Omega_2, \ldots, \Omega_\nu$

`Initialize:` $L = 0, \Lambda(x) = 1, \Lambda_{prev}(x) = 1,$

$\Omega(x) = 0, \Omega_{prev}(x) = 1, l = 1, d_m = 1$

```
for j = 1 to 2t
  d = S_j + Σ_{i∈{1..L}} Λ_i × S_{j−i}
  if (d = 0)
    l = l + 1
  else
    if (2L > j)
      Λ(x) = Λ(x) − d d_m^{−1} x^l Λ_prev(x)
      Ω(x) = Ω(x) − d d_m^{−1} x^l Ω_prev(x)
      l = l + 1
    else
      swap(Λ(x), Λ_prev(x))
      swap(Ω(x), Ω_prev(x))
      Λ(x) = Λ_prev(x) − d d_m^{−1} x^l Λ(x)
      Ω(x) = Ω_prev(x) − d d_m^{−1} x^l Ω(x)
      L = j − L
      d_m = d
      l = 1
```

Chien Search

The error locations are given by the inverse roots of $\Lambda(x)$, which are found using the Chien search algorithm. Error Locations loc_j are given by the following relation:

$$\Lambda(\alpha^{-loc_j}) = 0 \tag{6.5}$$

Pseudocode:

```
Input: Λ_1, Λ_2, ..., Λ_ν
Output: loc_1, loc_2, ..., loc_ν
```

```
for i = 0 to n
  Sum = 0
  for k = 0 to ν
    Sum = Sum + Λ_k α^{-ik}
  if (Sum = 0)
    loc_j = i
    j = j + 1
```

Forney's Algorithm

Using $\Lambda(x)$ and $\Omega(x)$, the error values are computed at the error locations. At all other indices the error values are zero. The magnitudes are found by Forney's algorithm which gives the following relation:

$$e_{loc_j} = -\frac{\Omega(\alpha^{-loc_j})}{\Lambda'(\alpha^{-loc_j})} \tag{6.6}$$

Pseudocode:

```
Input: Λ_1, ..., Λ_ν;   Ω_1, ..., Ω_ν;   loc_1, ..., loc_ν
Output: e_{loc_i}, ..., e_{loc_ν}
for i = 1 to ν
  for j = 0 to ν
    Sum_Ω = Sum_Ω + Ω_j α^{-loc_i j}
  for j = 0 to ν/2
    Sum_Λ = Sum_Λ + Λ_{2j-1} α^{-loc_i j}
  e_{loc_i} = Sum_Ω / Sum_Λ
```

Error Correction

The error locations and values obtained in previous steps are used in the correction of the received polynomial to produce the decoded polynomial.

$$d(x) = r(x) - e(x) \tag{6.7}$$

Pseudocode:

`Input:` $r_{n-1}, r_{n-2}, \ldots, r_{n-k-1}$; $e_{n-1}, e_{n-2}, \ldots, e_{n-k-1}$

`Output:` $d_{n-1}, d_{n-2}, \ldots, d_{n-k-1}$

`for` $i = n-1$ `to` $n-k-1$

$d_i = r_i - e_i$

It is straightforward to translate this pseudocode into actual code in almost any high level language.

6.3 Hardware Implementation

In this project, we first implemented the pseudocode presented in Section 6.2.2 in C++. This provided a golden functional model as well as solidified our understanding of the algorithm. After we had a working reference implementation, we used the following approach to implement the design in the C-based tool and Bluespec.

1. Define the top level structure and module interfaces of the design
2. Implement each module by translating the C++ reference into target language
3. Iteratively refine each module to meet the design constraints

6.3.1 Defining Module Interfaces

We first define the top level architecture of the hardware design, which describes the module partitioning and the inter-module communications. A language which

decouples interface from implementation facilitates modular refinements, which are important for design space exploration. We next discuss how the C-based synthesis tool and Bluespec achieve this goal.

C-based Synthesis: Each hardware module can be declared as a function that takes inputs and gives outputs using pointers. A hierarchy of modules can be formed by writing a function which calls other functions. Data communication between modules in the same hierarchy is automatically inferred by the C-based tool's compiler by data dependency analysis. Without modifying the function declarations, Our C-based tool allows system designers to explicitly specify the widths and the implementation types of data communication channels. The available implementation types include wires, FIFOs or RAMs.

A function call can be considered as a transaction. The C-based synthesis tool's compiler utilizes well known compilation techniques to exploit parallelism. For example, it can exploit the data parallelism within the function by loop unrolling and data flow analysis. It can also exploit pipelining across consecutive calls of the same function through software pipelining and parallel executions of modules in the same hierarchy. While these techniques are efficient to some extent, they have their limitations. For example, loop unrolling and software pipelining are only applicable to loops with statically determined number of iterations. The transaction granularity on which the function operates is a tradeoff between performance and implementation effort. Coarse-grained interface, in which each function call processes a large block of data, allows system designers to naturally implement the algorithm in a fashion similar to software. On the other hand, fine-grained interface allows the compiler to exploit fine-grained parallelism existing in the algorithm at the expense of higher implementation effort. Similar observations about these two styles are mentioned in [34], where coarse-grained functions are referred to as *block mode processing* and fine-grained functions as *throughput mode processing.*

In our design, we represent each stage of the Reed-Solomon decoder as a separate function. This naturally translates into block mode processing with coarse grained interfaces between modules. Making the interfaces fine-grained would have greatly

increased code complexity as discussed in Section 6.5 and reduced the modular nature of the design.

Bluespec: As a hardware description language, Bluespec inherently has the concept of modules. A module can communicate with external logic only through methods. Unlike Verilog/VHDL ports which are wires, Bluespec methods are semantically similar to methods in object oriented programming languages, except that they also include associated ready/enable handshake signals. This facilitates implementation of latency insensitive modules. A module interface is declared separately from its implementation. Since Bluespec supports polymorphism, a polymorphic interface can be used to control the communication channel width. However, this requires the implementation of the module to also be polymorphic, which in general increases the implementation effort.

The high-level dataflow of the Reed-Solomon decoding algorithm can be implemented using FIFOs to achieve a latency insensitive pipeline as shown in Figure 6-4. Each module's interface simply consists of methods to enqueue and dequeue data with underlying Bluespec semantics taking care of full and empty FIFOs. This interface can be further parameterized, with some programming effort but no hardware penalty, by the number of the elements to be processed in parallel.

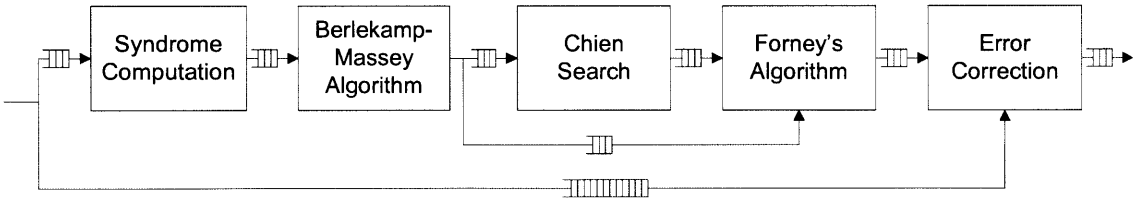


Figure 6-4: Bluespec interface for the decoder

6.3.2 Initial C-based Implementation

The C-based synthesis tool emphasizes the use of for-loops to express repeated computations within a module. To expose parallelism it provides the user with knobs to determine loop unrolling or loop pipelining. As previously discussed in Section 6.2.2, each of the major computational steps of Reed-Solomon decoding have this structure.

Thus we expect the tool to be very effective at providing a direct implementation of the decoding steps in terms of modules, each having for-loops of different lengths. Since the tool has the ability to compile native C/C++ program, obtaining the initial synthesized design is trivial. The compiler automatically designs the finite state machines (FSMs) associated with each module. The pseudocode described in Section 6.2.2 was modified to the C++ subset accepted by the tool. Memory allocation and deallocation is not supported and pointers have to be statically determined. The for-loops were kept as described in the pseudocode; no loop unrolling was performed. The resulting hardware took on 7.565 million cycles per input block, for the worst case error scenario. The high cycle count in the C-based implementation was due to the tool generating a highly sequential implementation. To improve the throughput, we made use of the techniques offered by the tool.

Loop Unrolling: The C-based tool's compiler can automatically identify loops that can be unrolled. By adding annotations to the source code, the user can specify which of these identified loops need to be unrolled and how many times they will be unrolled. As a first step we unrolled the loops corresponding to the GF Multiplication (Section 6.2.2), which is used multiple times throughout the design. Next, the inner for-loop of Syndrome computation (Section 6.2.2) was unrolled to parallelize the block. The inner for-loop of the Chien search (Section 6.2.2) was also unrolled. To perform these steps we had to replace the dynamic parameters t and ν by static upper bounds. These unrolling steps lead to a significant improvement in the throughput, achieving 19020 cycles per input block, as shown in Figure 6-5. This is 7% of the target data throughput.

Unrolled functions	**Throughput(Cycles/Block)**
None	7565K
GF Mult	237K
GF Mult, Syndrome	33K
GF Mult, Syndrome, Chien	19K

Figure 6-5: Performance impact of unrolling

The unrolling steps lead to a simplification of the complex FSMs initially generated at the cost of increased arithmetic operations per module. To further improve the

performance, we needed to achieve further pipelining of some blocks, which required complex refinements discussed in Section 6.5.

6.3.3 Initial Implementation in Bluespec

We generated an initial implementation of the decoder by manually translating each for-loop into a simple FSM where each cycle executes a single loop body. The control signals are generated by the compiler automatically. The code shown in Figure 6-6 illustrates how the pseudocode of Syndrome Computation given in Section 6.2.2 was translated into a Bluespec module.

```
module mkSyndrome (Syndrome);
  FIFO#(Byte) r_in_q <- mkLFIFO();
  FIFO#(Vector#(32,Byte)) s_out_q <- mkLFIFO();
  Reg#(Vector#(32,Byte)) syn <- mkReg(replicate(0));
  Reg#(Byte) i <- mkReg(0);
  Reg#(Byte) j <- mkReg(0);

  rule compute_syndrome (True);
    let new_syn = syn;
    let product = gf_mult(new_syn[j],alpha(j+1));
    new_syn[j]  = gf_add(r_in_q.first(), product);

    if (j + 1 >= 2*t)
      j <= 0;
      r_in_q.deq();
      if (i + 1 == n)
        s_out_q.enq(new_syn);
        syn <= replicate(0);
        i <= 0;
      else
        i <= i + 1;
    else
      syn <= new_syn;
      j <= j + 1;
  endrule

  method r_in(r_data) = r_in_q.enq(r_data);
  method s_out();
    s_out_q.deq();
    return s_out_q.first();
  endmethod
endmodule
```

Figure 6-6: Initial version of syndrome module

The input and output of the module are buffered by two FIFOs, *r_in_q* and *s_out_q*. These FIFOs are one-element Bluespec library FIFOs (*mkLFIFO*) which allow concurrent enqueue and dequeue. We also instantiated three registers: *syn*, *i* and *j*. *syn* stores the temporary values of the syndrome. The last two are used for loop bookkeeping. The entire FSM is represented by a single rule called *compute_syndrome* in the module. This rule models the semantics of the two nested for-loops presented in Section 6.2.2. The code in bold corresponds to the loop body computation that appeared in the pseudocode. The remaining code describes how the registers and FIFOs are updated with the appropriate computation results and bookkeeping values. The GF arithmetic operations, *gf_mult* and *gf_add*, are implemented as library functions which are compiled into combinational logic.

We implemented each of the five modules using this approach. For $t = 16$ (32 parity bytes), we obtained a throughput of 8161 cycles per data block, i.e. the decoder could accept a new incoming data block every 8161 cycles. This is 17% of the target data throughput. It should be noted that even in this early implementation, computations in different modules already overlap due to the FIFOs' decoupling.

6.4 Design Refinements in Bluespec

After the initial implementation, our next step was to make incremental refinements to the design to improve the performance in terms of reducing the number of cycles taken to process one input block.

Assuming the input data stream is available at the rate of one byte per cycle, for the best performance the decoder hardware should accept one input byte per clock cycle giving a throughput close to 255 cycles per input block. In the next few paragraphs, we describe what refinements are needed to achieve this and how the Bluespec code needed to be modified.

Manual Unrolling: Unlike C-based synthesis tools, Bluespec requires users to explicitly express the level of parallelism they want to achieve. That means source code modification may be needed for different implementations of the same algorithm.

However, the transformations usually do not require substantial change of the source code. In some occasions, user may even be able to use static parameterization to model the general transformations such as loop unrolling. We illustrate this using the Syndrome Computation module. This module requires $2t$ GF Mults and $2t$ GF Adds per input symbol, which can be performed in parallel. Our initial implementation only performs one multiplication and one addition per cycle. By modifying the rule as shown in Figure 6-7, the module can complete **par** times the number of operations per cycle.

```
rule compute_syndrome (True);
  let new_syn = syn;
  for (Byte p = 0; p < par; p = p + 1)
    let product = gf_mult(in_q.first, alpha(i+p+1));
    new_syn[i+p] = gf_add(new_syn[i+p], product);

  if (j + par >= 2*t)
    j <= 0;
    in_q.deq();
    if (i + 1 == n)
      out_q.enq(new_syn);
      syn <= replicate(0);
      i <= 0;
    else
      i <= i + 1;
  else
    syn <= new_syn;
    j <= j + par;
endrule
```

Figure 6-7: Parameterized parallel version of the compute-syndrome rule

As seen above, the code is nearly identical to the original with the modifications highlighted in bold. The only change is the addition of a user specified static variable **par** which controls the number of multiplications and additions the design executes per cycle.

We unrolled the computations of the other modules using this technique, which made the design able to accept a block every 483 cycles. At this point, the design throughput was already 315% of the target performance. To achieve the ultimate throughput of accepting a block per 255 cycles, we found that the Forney's algorithm module was the bottleneck.

Out-of-order Execution: Figure 6-8 shows a simplified view of the implementation of the Forney's algorithm module.

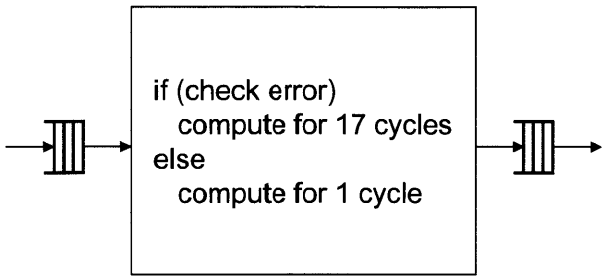


Figure 6-8: The original structure of the Forney's Algorithm Implementation

This implementation processes input symbols in the order of their arrivals. Depending on the value of the symbol, the module takes either 17 cycles or one cycle to process it. As it is assured that at most t symbols will need to be processed for 17 cycles in a data block of size k, our non-pipelined design can handle a data block every $17t + (k - t) = k + 16t$ cycles in the worst case scenario. For example, if $t = 16$ and $k = 223$, the design can only accept a new block every 479 cycles.

Data flow analysis shows that there is no data dependency between the computations of the input elements. The only restriction is that results must be forwarded to the next module in the input order. Our original architecture does not take advantage of this characteristic because the reference C code implies in-order executions. To increase the throughput, we modified the architecture of our design to support out-of-order executions. We split the module into four sub-modules connected by FIFOs as shown in Figure 6-9. In the new design, every input is first passed to a submodule called *check input*. This module then issues the element to its corresponding processing unit according to its value, and provides *merge* with the correct order for retrieving data from these processing units. The new design is able to handle a data block every $max(17t, k - t)$ cycles, this translates to a throughput number of one block per 272 cycles.

Buffer Sizing: The sizes of FIFO buffers in the system have a large impact on

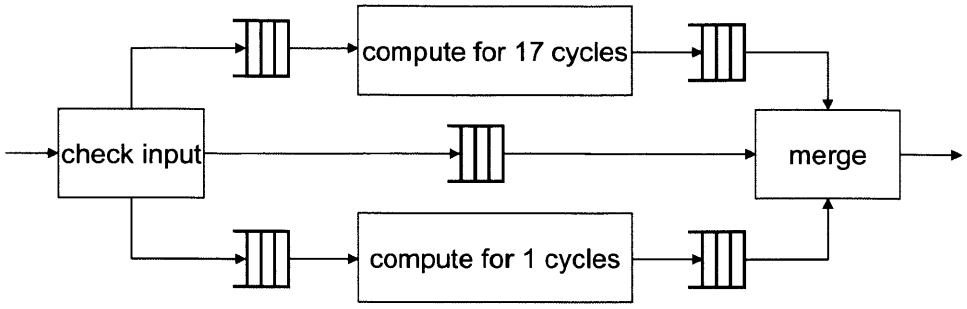


Figure 6-9: The modified structure of the Forney's Algorithm Implementation

the system throughput. In a complex system consisting of modules with varying data rates, it is difficult to calculate the optimal size of each buffer. A common method to compute buffer sizes is through design exploration with cycle accurate simulations. It is trivial to adjust the sizes of the FIFOs with the Bluespec library. Figure 6-10 shows the performance impact of the sizing of the largest FIFO, which connects the input data port with the Error Correction module. We can see from the result that the design achieves maximum throughput when the buffer is large enough to store at least three 255 byte input data blocks.

Buffer Size	**Throughput (Cycle/Block)**
255	622
510	298
765	276
1020	276
1275	276

Figure 6-10: Performance impact of FIFO size when $n = 255, t = 16$

Synthesis of this design with a buffer size of 765 bytes showed that the throughput of 276 cycles per input block is sufficient for our requirement, as seen in Section 6.6.

6.5 Language-related issues with refinements

Generating hardware for streaming DSP applications requires some more refinements. We now describe these in context of both Bluespec and C-based design methodologies

and the issues related to the same.

Pipelining: To further improve throughput, the decoder was set up as a pipeline with streaming data blocks. For hardware efficiency, consecutive modules should be able to share partially computed data to have overlapping execution on a data block. For example, once the Chien search module determines a particular error location, that location can be forwarded immediately to the Forney's algorithm module for computation of the error magnitude, without waiting for the rest of error locations to be determined. Another requirement for high performance is to have the pipeline stages balanced, with each module processing its input in the same number of cycles.

Both these requirements could be satisfied in Bluespec by the use of explicitly sized FIFO buffers. In the above error location example, having a location vector FIFO between the modules, allowed for significant overlapping execution between the modules. However, representing this requirement is unnatural in the C-based tool due to presence of dynamic length for-loops in the design. This makes the task of generating the inter-module buffers quite tedious.

Problems with Streaming under Dynamic Conditions: In the C-based tool separate modules (represented as different functions) share data using array pointers passed as arguments. For simple streaming applications, the compiler can infer that both the producer and consumer operate on data symbols in-order and can generate streaming hardware as expected by the designer. To illustrate how this happens more clearly, consider the code segment shown in Figure 6-11.

```
void producer(char input[255],
              char intermediate[255])
{
  for (int i=0; i<255; i++)
    intermediate[i]=input[i]+i;
}
void consumer(char intermediate[255],
              char output[255])
{
  for (int i=0; i<255; i++)
    output[i]=intermediate[i]-i;
}
```

Figure 6-11: Simple Streaming Example

The C-based tool's compiler correctly generates streaming hardware in the form shown in Figure 6-12. Here the compiler infers that a pipe length of 8 bytes was optimal.

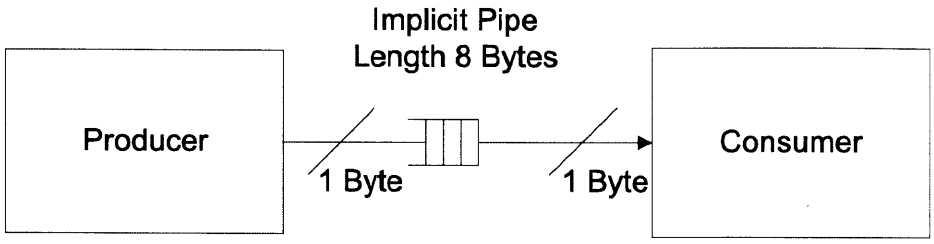


Figure 6-12: Simple streaming example

The presence of dynamic parameters in for-loop bounds, however, can obfuscate the sharing of streamed data. If the data block is conditionally accessed, these dynamic conditions prevent the compiler from inferring streaming architecture. For example, consider the code segment shown in Figure 6-13, where the producer dynamically determines the data length on which to operate and produces values sporadically.

```
void producer(char input[255], char length,
              char intermediate[255], char *count)
{
  *count = 0;
  for (int i=0; i<length; i++)
    if (input[i]==0)
      intermediate[(*count)++]=input[i]+i;
}
void consumer(char intermediate[255], char *count,
              char output[255])
{
  for (int i=0; i<*count; i++)
    output[i]=intermediate[i]-i;
}
```

Figure 6-13: Complex Streaming Example

This results in the hardware shown in Figure 6-14. The compiler generates a large RAM for sharing one instance of the *intermediate* array between the modules. To

ensure the program semantics, the two modules will not be able to simultaneously access the array, which prevents overlapping of execution of the two modules.

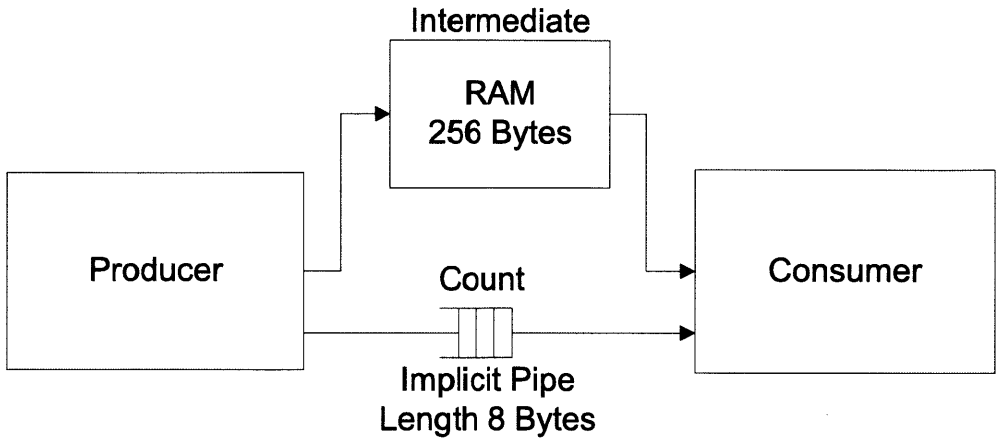


Figure 6-14: Complex Streaming example

The C-based synthesis tool provides an alternative buffer called ping-pong memory which uses a double buffering technique, to allow some overlapping execution, but at the cost of extra hardware resources. Using this buffer, our design's throughput improved to 16,638 cycles per data block. The cycle count was still high due to the complex loops in Berlekamp module.

Substantial Source Code Modifications: For further improving the performance and synthesis results, we made use of common guidelines [81] for code refinement. Adding hierarchy to Berlekamp computations and making its complex loops static by remove the dynamic variables from the loop bounds, required algorithmic modifications to ensure data consistency. By doing so, we could unroll the Berlekamp module to obtain a throughput of 2073 cycles per block, which was close to the minimum requirement. However, as seen in Section 6.6, the synthesized hardware required considerably more FPGA resources than the other designs. Further optimizations require expressing module functions in a fine-grained manner, i.e. operating on a symbol-by-symbol basis. This leads to considerable complexity as hierarchically higher modules have to keep track of individual symbol accesses within a block. The modular design would need to be flattened completely, so that a global FSM can

be made aware of fine-grained parallelism across the design. Due to this increased code complexity, the abstraction provided by a sequential high level language is broken. Moreover, it is difficult for algorithm designers to express the inherent structure associated with such designs in C/C++. Others have expressed similar views [25]. This forms the basis for the inefficiency in generated hardware which we encountered during our study.

6.6 Results

The total number of source code lines of the final Bluespec design add up to 1759 lines. On the other hand, source code of the C-based version of the design takes 956 lines and the associated constraints file with the user-specified directives has 90 lines. The increased size of Bluespec code is primarily due to the explicit module interfaces and the associated methods in Bluespec. For comparison, an open source reference RTL implementation [59] has non-parameterized Verilog source code with 3055 lines for $n = 255$ and $t = 16$. Figure 6-15 shows the FPGA synthesis summary of the final designs developed using the two design flows. We also include the Xilinx IP core as a baseline reference. We can see that the Bluespec design achieves 178% of the Xilinx IP's data rate with 90% of the equivalent gate count. On the other hand, the C-based design, achieves only 23% of the IP's data rate with 200% of the equivalent gate count.

6.7 Summary

Through this study we show that even for highly algorithmic code with relatively simple modular structure, architectural issues dominate in determining the quality of hardware generated. If the primary goal of the design is to speedily generate hardware for prototyping or FPGA based implementations without an emphasis on the amount of resources and fine tuned performance, then C-based design offers a shorter design time provided the generated design fits in the FPGA. On the other hand, high-

Design	Bluespec	C-based	Xilinx
LUTs	5863	29549	2067
FFs	3162	8324	1386
Block RAMs	3	5	4
Equivalent Gate Count	267,741	596,730	297,409
Frequency (MHz)	108.5	91.2	145.3
Throughput (Cycles/Block)	276	2073	660
Data rate (Mbps)	701.3	89.7	392.8

Figure 6-15: FPGA synthesis summary of the three IPs for $n = 255, t = 16$. The Bluespec design is generated by Bluespec Compiler 2007.08.B. The C-based design is generated by the verion released at the same time frame as the Bluespec's compiler. The Xilinx Reed-Solomon Decoder IP is version 5.1. All designs are compiled by Xilinx ISE version 8.2.03i with Virtex-II Pro as the target device.

level HDL languages offer better tools for designing hardware under performance and resource constraints while keeping the benefits of high level abstraction. Insight about hardware architecture like resource constraints, modular dataflow, streaming nature and structured memory accesses can greatly improve the synthesized design. Algorithm designers need intuitive constructs to exert control over these issues. This is difficult to express in languages like C/C++, while hardware-oriented languages like Bluespec offer well defined semantics for such design. Because of this reason, we opt for Bluespec as our design language for developing hardware designs for our Airblue platform.

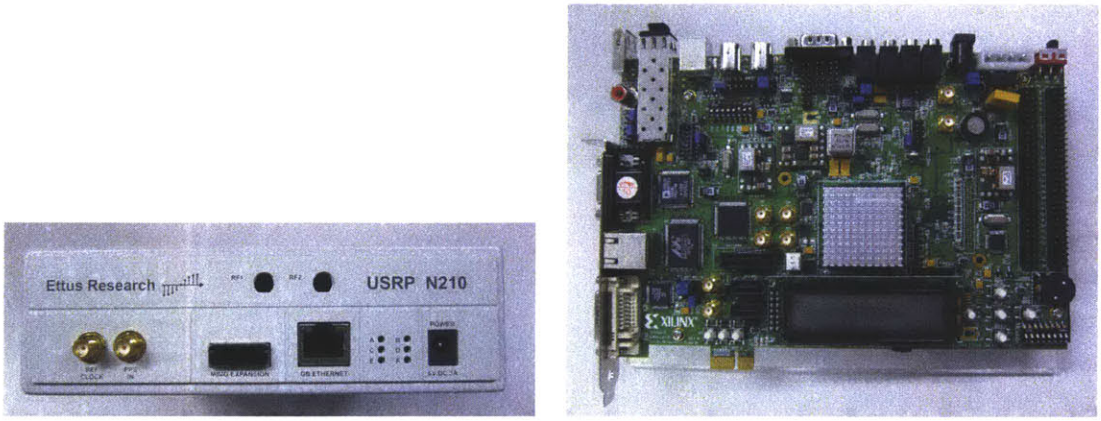
Chapter 7

Conclusion And Future Work

Future wireless protocols require new features and functions in various layers of the networking stack, a systematic way of passing additional information from lower layers to higher layers, and a fast way of controlling lower layers from higher layers. Simulations with channel models are not adequate to evaluate these protocols. Therefore, a prototyping platform that is easy to modify and can achieve 802.11-like throughput and latency is needed. The main contribution of this thesis is the development of Airblue, which is a platform implemented with this goal in mind.

In this thesis, we have shown that a modifiable protocol must be implemented in a *latency-insensitive* manner and must *pass control in a data-driven* manner. We have also explained how streaming interfaces between the PHY and MAC help in lowering the processing latency. Moreover, we have demonstrated that highly reusable modules can be developed using static or dynamic parameterizations. Static parameterizations are useful for architectural explorations. Dynamic parameterizations provide the flexibility to implement adaptive protocols. Through the implementation of SoftRate, we have demonstrated that Airblue is easy to modify and that, when modified, it meets the performance requirements of current wireless protocols. In particular, we can easily modify the platform to send per-packet feedback, implement new decoding algorithms, and perform runtime reconfigurations of the pipeline, all while meeting 802.11 timing requirements.

We would like to make Airblue publicly available for other researchers. Unfortu-



(a) USRP2 from Ettus Research

(b) XUPV5 from Xilinx

Figure 7-1: The new platform for Airblue. The platform consists of two FPGA boards: USRP2 and XUPV5. The former behaves as the RF frontend. It communicates baseband signals with the latter through a high-speed bi-directional serial link. The Airblue's baseband and MAC processing are implemented on the XUPV5. Upper layers are implemented on a host PC connecting to the XUPV5 through PCIe.

nately, Nokia no longer manufactures the FPGA boards on which our original Airblue can run. As a result, we are in the processing of porting our Airblue implementation to a new FPGA platform as shown in Figure 7-1. The new platform consists of two FPGA boards: 1) USRP2 from Ettus Research and 2) XUPV5 from Xilinx. USRP2 is capable of communicating baseband signals at sampling rates up to 100 MHz to a PC through a high-speed bi-directional serial link. Moreover, by plugging in different daughter boards with different radios, USRP2 can support different ranges of carrier frequencies. Because of these capabilities, USRP2 is often used with the GNURadio software to form software-based radios. In our new system, we connect a USRP2 to another FPGA called XUPV5 with the high-speed bi-directional serial link. We implement Airblue baseband and MAC processing on XUPV5. Other higher layers are implemented on a host PC connecting to XUPV5 through PCIe. We are now packaging up our code for a summer 2011 release. The main advantage of this new setup is that both USRP2 and XUPV5 are commodity products that are well supported by the companies developing them. Moreover, research groups in the academia can acquire them at discounted prices with the total cost of ownership being less than 2000 US dollars per node.

7.1 New Experimentation Ideas

We have already discussed in the thesis how Airblue can be used to implement existing cross-layer protocol like SoftRate. In the near future, we have two additional ideas we would like to explore using the Airblue platform.

7.1.1 Development and Evaluation of Architectural Power Saving Measures

Current 802.11 receivers decode all the packets they received, even if the packets are not intended for them. While doing so does not affect the correctness of the protocol, it is not ideal in terms of power consumption. Substantial power savings can be achieved if the receiver powers down once it finds out that the packet is not meant for it. Ideally, the receiver should shutdown as early as possible and wake up right after the end of the packet so that it will not miss future packets. We believe Airblue is an ideal platform to evaluate different implementations of these power-saving measures. For example, one way of deciding whether to reject a packet is to check the MAC address of the destination. Such measure only makes sense if the MAC address can be inspected way before the end of the packet. Airblue's streaming interface achieves exactly this. Another proposal is to make the decision even earlier by making use of the channel properties measured by the synchronizer and the channel estimator to identify the source node of the packet. This requires us to modify the baseband pipeline to expose this information to the MAC which Airblue has been shown to achieve effectively.

7.1.2 Development of Empirical Channel Models

As we mentioned in Chapter 1, the main problem of protocol evaluation using simulations is that it is hard to accurately model the channel because the channel varies with the actual protocol. On the other hand, on-air experimentations have the problem of being hard to verify because of lack of repeatability. We plan to use the Airblue

infrastructure to derive an empirical channel model to solve this problem. First, we will estimate the values of various channel parameters such as signal-to-noise-ratio (SNR), carrier frequency offsets (CFO), subcarrier attenuation and phase offsets with different set of nodes interfering with each other. Then, we measure the timing characteristic of the protocol by using the cycle-accurate simulator. After that, the empirical channel model will be derived by combining the timing and channel characteristics measured. We believe that such a model can potentially have the benefits of both simulations and on-air experimentations if it is properly constructed.

7.2 Airblue Extensions

While the current Airblue is a very powerful platform for protocol experimentations, we plan to improve the platform in term of usability and features supported.

7.2.1 Usability Enhancements

Although the current Airblue implementation is very modifiable, one needs to learn Bluespec in order to modify Airblue. This requirement may become the entry barrier for protocol designers who are not familiar with hardware designs. We would like to lower the barrier by improving Airblue's Graphical User Interface (GUI). The goal is to enable users to understand the design and make controlled modifications via Application Programming Interfaces (APIs) provided by the GUI. Airblue's current design management tool AWB has some supports to achieve our goal. To understand Airblue design, users can see the top-level structure of Airblue via AWB's GUI. However, we would like to extend to GUI so that it will also show the module connections, interfaces and data types. To modify Airblue, AWB's plug-and-play support allows user to pick the desired implementation for a module from a list of available implementations. In addition, the values of many static parameters can be set in AWB GUI without modifying the underlying source code written in Bluespec. In the future, we plan to add more APIs to AWB to ease Airblue modification. One idea is to develop a tool which can automatically generate controllers that configure

the Airblue pipelines. Another idea is to allow users to add debug assertions to any connection in the design using the GUI.

7.2.2 Hardware Extension

All protocol extensions described in this thesis involve only modifying the program being run on the FPGA. However, some experimental protocols may require additional hardware support that is not supported by Airblue currently. A good example is Multiple-Input-Multiple-Output (MIMO), which increases data throughput via the usage of multiple antennas. The following discusses how Airblue can support MIMO in the future.

Support of Large-Scale Experimentation

Our current setup requires each XUPV5 to be connected to a remote PC through PCI-E. We see this as a limitation to carry out large-scale experiments. There are two extensions that can tackle this limitation. First, we plan to use the Gb-Ethernet instead of PCI-E for communication with the remote PC. By making each XUPV5 appear as an individual node in the ethernet network, a single PC can configure multiple XUPV5 devices remotely through the ethernet network. In this scenario, an experimental node is only required to connect to a close by ethernet socket, getting rid of the requirement of being physically attached to a PC. Second, we plan to use the SD card reader of XUPV5 to store experimental results and process them offline after the experiment. In this case, after the XUPV5 boards are configured, they can act as standalone devices for experiments which further facilitate large-scale experimentation.

Support of MIMO

Currently, Airblue does not support MIMO because we believe that the FPGA on XUPV5 is not large enough to host a MIMO implementation (our non-MIMO implementation already uses over 70% of the FPGA resources on XUPV5). With the

right FPGA board which hosts a large FPGA and multiple high-speed serial link connectors, we believe we can extend Airblue to support MIMO by connecting multiple USRP2 to the board. We can apply the same virtualization technique to abstract the MIMO radio interface that is visible by the baseband. Finally, we can modify the baseband to deal with the additional input and output streams required by MIMO. Our understanding is that most modules except the channel estimator can be reused directly from our non-MIMO implementation. Figure 7-2 shows the proposed MIMO platform for Airblue with 2 inputs and 2 outputs.

If the extensions discussed in this section are implemented, Airblue will become even more powerful in wireless protocol development.

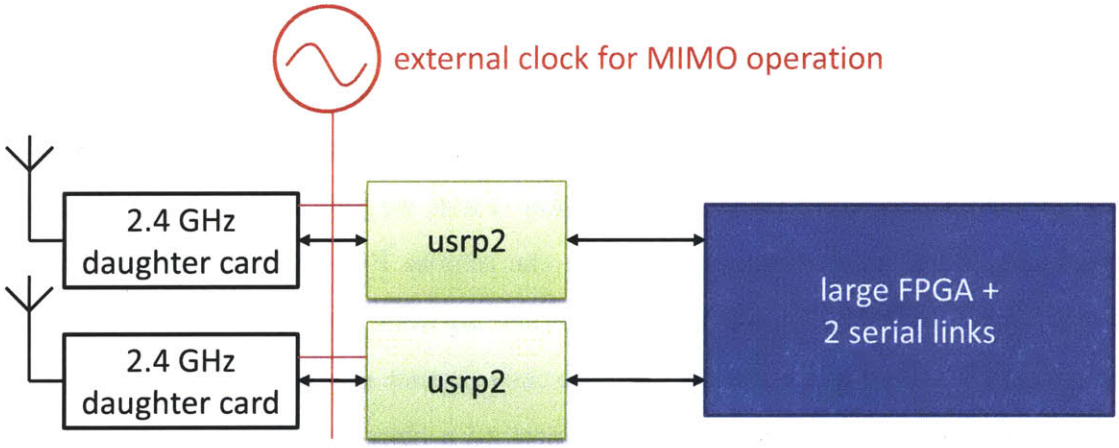


Figure 7-2: Proposed 2x2 MIMO platform for Airblue: 2 USRP2 are connected to an advanced FPGA board with multiple serial link connectors. An external clock is used to drive all the USRP2s and the daughter cards to ensure synchronized clock for MIMO operation. Baseband implemented on the FPGA will need to be modified to deal with the additional stream of IQs.

Bibliography

[1] Altera DSP Builder. `http://www.altera.com/products/prd-index.html`.

[2] Engling Yeo Stephanie Augsburger, Wm. Rhett Davis, and Borivoje Nikolic. 500 Mb/s Soft Output Viterbi Decoder. In *ESSCIRC'02*.

[3] Architect's Workbench. `http://asim.csail.mit.edu/redmine/projects/show/awb`.

[4] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE TIT*, 20(2), 1974.

[5] Prithviraj Banerjee, Malay Haldar, Anshuman Nayak, Victor Kim, Vikram Saxena, Steven Parkes, Debabrata Bagchi, Satrajit Pal, Nikhil Tripathi, David Zaretsky, Robert Anderson, and Juan Ramon Uribe. Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs. *IEEE Transactions of Very Large Scale Integration (VLSI) Systems*, 12(3), 2004.

[6] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara. Soft-Output Decoding Algorithms for Continuous Decoding of Parallel Concatenated Convolutional Codes. In *ICC'96*.

[7] Vaclav E. Benes. Mathematical Theory of Connecting Networks and Telephone Traffic. *Academic Press*, 1965.

[8] Elwyn R. Berlekamp. Nonbinary BCH Decoding. In *Proceedings of Internation Symposium on Information Theory*, San Remo, Italy, 1967.

[9] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding. In *ICC'93*.

[10] Claude Berrou, Patrick Adde, Ettiboua Angui, and Stephane Faudeil. A Low Complexity Soft-Output Viterbi Decoder Architecture. In *ICC'93*.

[11] Bluespec Inc. http://www.bluespec.com.

[12] David Brier and Raj Mitra. Use of C/C++ models for architecture exploration and verification of DSPs. In *DAC '06*.

[13] Cadence. C-to-Silicon Compiler. `www.cadence.com`.

[14] Celoxica. Handel-C. http://www.celoxica.com/technology/.

[15] Ranveer Chandra, Ratul Mahajan, Thomas Moscibroda, Ramya Raghavendra, and Paramvir Bahl. A case for adapting channel width in wireless networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[16] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[17] Robert W. Chang. Synthesis of band-limited orthogonal signals for multichannel data transmission. *Bell System Technical Journal*, 45, 1966.

[18] R. T. Chien. Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. *IEEE Transactions on Information Theory*, 10, 1964.

[19] James Cooley and John Tukey. An Algorithm for The Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.

[20] Nirav Dave. Designing a Processor in Bluespec. Master's thesis, MIT, Cambridge, MA, Jan 2005.

[21] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. Modular Refinement and Unit Testing. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Grenoble, France, 2010.

[22] Nirav Dave, Michael Pellauer, Steve Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Napa, CA, 2006.

[23] Angela Doufexi, Simon Armour, Peter Karlsson, Andrew Nix, and David Bull. A Comparison of the HIPERLAN/2 and IEEE 802.11a Wireless LAN Standards. *IEEE Commun. Mag.*, 40, 2002.

[24] Farinaz Edalat, Jit Ken Tan, Khoa M. Nguyen, Nir Matalon, and Charles G. Sodini. Measured Data Rate from Adaptive Modulation in Wideband OFDM Systems. In *Proceedings of IEEE International Conference on Ultra Wide Band*, Waltham, MA, 2006.

[25] S. A. Edwards. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers*, 23(5):375–386, May 2006.

[26] Kermin Elliott Fleming, Man Cheuk Ng, Samuel Gross, and Arvind. WiLIS: Architectural Modeling of Wireless Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, 2011.

[27] G. D. Forney. On Decoding BCH Codes. *IEEE Transactions on Information Theory*, 11, 1965.

[28] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer, 2000.

[29] The GNURadio Software Radio. `http://gnuradio.org/trac`.

[30] Shyamnath Gollakota and Dina Katabi. ZigZag decoding: Combating hidden terminals in wireless networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[31] T Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*, chapter 8. Springer, 2002.

[32] Ramakrishna Gummadi and Hari Balakrishnan. Wireless Networks Should Spread Spectrum Based On Demands. In *Hotnets-VII*, Calgary, Canada, 2008.

[33] Ramakrishna Gummadi, Rabin Patra, Hari Balakrishnan, and Eric Brewer. Interference avoidance and control. In *Hotnets-VII*, Calgary, Canada, 2008.

[34] Y. Guo, D. McCain, J. R. Cavallaro, and A. Takach. Rapid Prototyping and SoC Design of 3G/4G Wireless Systems Using an HLS Methodology. *EURASIP Journal on Embedded Systems*, 2006(1):18–18, 2006.

[35] Joachim Hagenauer and Peter Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *GLOBECOM'89.*

[36] Daniel Halperin, Josephine Ammer, Thomas Anderson, and David Wetherall. Interference Cancellation: Better receivers for a new Wireless MAC. In *Hotnets-VI*, 2007.

[37] Daniel Halperin, Thomas Anderson, and David Wetherall. Taking the Sting out of Carrier Sense: Interference Cancellation for Wireless LANs. In *MobiCom'08*, San Francisco, CA, 2008.

[38] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *ICCAD'00*, pages 511–518, San Jose, CA, 2000.

[39] Jeff Hoffman, David Arditti Ilitzky, Anthony Chun, and Aliaksei Chapyzhenka. Architecture of the Scalable Communication Core. In *NOCS'07*, Princeton, NJ, 2007.

[40] G. Holland, N. Vaidya, and P. Bahl. A Rate-Adaptive MAC Protocol for Multihop Wireless Networks. In *MOBICOM'01*, Rome, Italy, 2001.

[41] Ramsey Hourani, Ravi Jenkal, Rhett Davis, and Winser Alexander. Automated Architectural Exploration for Signal Processing Algorithms. In *ASILOMAR Conf. on Signals, Systems and Computers*, 2006.

[42] http://www.nallatech.com. Nallatech acp module.

[43] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[44] IEEE. *IEEE standard 802.16. Air Interface for Fixed Broadband Wireless Access Systems*, 2004.

[45] Kyle Jamieson. *The SoftPHY Abstraction: from Packets to Symbols in Wireless Network Design.* PhD thesis, MIT, Cambridge, MA, 2008.

[46] Kyle Jamieson and Hari Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *SIGCOMM'07.*

[47] Wen Ji, Yiqiang Chen, Min Chen, and Yi Kang. Unequal error protection based on objective video evaluation model. In *MobiMedia'07*, Nafpaktos, Greece, 2007.

[48] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura. A C-based synthesis system, Bach, and its application. In *Proceedings of ASP-DAC*, 2001.

[49] Sachin Katti, Shyamnath Gollakota, and Dina Katabi. Embracing wireless interference: analog network coding. In *SIGCOMM'07*, Kyoto, Japan, 2007.

[50] Sachin Katti, Dina Katabi, Hari Balakrishnan, and Muriel Medard. Symbol-Level Network Coding for Wireless Mesh Networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[51] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. XORs in the air: practical wireless network coding. In *SIGCOMM '06: Proceedings of the ACM SIGCOMM 2006 Conference on Data communication*, 2006.

[52] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM TCS*, 18(3), 2000.

[53] Milos Krstic, Koushik Maharatna, Alfonso Troya, Eckhard Grass, and Ulrich Jagdhold. Baseband Processor for IEEE 802.11a Standard with Embedded BIST. *Facta Universitatis, Series: Electronics and Energetics*, 17, 2004.

[54] Lang Lin and Roger S. Cheng. Improvements in SOVA-Based Decoding For Turbo Codes. In *ICC'97*.

[55] G. Masera, G. Piccinini, M. Roch, and M. Zamboni. VLSI Architectures for Turbo Codes. *IEEE Trans. on VLSI Systems*, 1999.

[56] James L. Massey. Shift-Register Synthesis and BCH Decoding. *IEEE Transactions on Information Theory*, 15(1), 1969.

[57] Robert J. Mceliece. On the bcjr trellis for linear block codes. *IEEE Trans. Inform. Theory*, 1996.

[58] Mentor Graphics. Catapult-C. http://www.mentor.com/products/esl/.

[59] Ming-Han Lei. Reference Verilog Implementation. http://www.humanistic.org/ hendrik/reed-solomon/index.html.

[60] Todd K. Moon. *Error Correction Coding-mathematical methods and Algorithms.* Wiley-Interscience, New York, 2005.

[61] Thomas Moscibroda, Ranveer Chandra, Yunnan Wu, Sudipta Sengupta, and Paramvir Bahl. Load-aware spectrum distribution in wireless LANs. In *IEEE ICNP*, 2008.

[62] Man Cheuk Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communication Systems (ANCS)*, San Diego, CA, 2010.

[63] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse

Across Different OFDM Protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*.

[64] Grace Nordin, Peter Milder, James Hoe, and Markus Puschel. Automatic Generation of Customized Discrete Fourier Transform IPs. In *DAC'05*.

[65] George Nychis, Thibaud Hottelier, Zhuochen Yang, Srinivasan Seshan, and Peter Steenkiste. Enabling MAC Protocol Implementations on Software-defined Radios. In *NSDI'09*, Boston, MA, 2009.

[66] Open SystemC Initiative. SystemC language. `www.systemc.org`.

[67] David W. Page and LuVeme R. Peterson. Re-programmable pla. US Patent 4508977, Burroughs Corporation, 1983.

[68] Angshuman Parashar, Micheal Adler, Micheal Pellauer, Kermin Fleming, and Joel Emer. Leap: An operating system for fpgas. 2010.

[69] Michael Pellauer, Michael Adler, Derek Chiou, and Joel Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC'09*, San Francisco, CA, 2009.

[70] Picochip. `http://www.picochip.com`.

[71] Teemu Pitkanen, Vesa-Matti Hrtikainen, Nirav Dave, and Gopal Raghavan. 802.15.3 Transmitter: A Fast Design Cycle Using the OFDM Framework in Bluespec. In *Proceedings of International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS)*, Samos, Greece, 2008.

[72] Hariharan Rahul, Farinaz Edalat, Dina Katabi, and Charles Sodini. Frequency-Aware Rate Adaptation and MAC Protocols. In *MobiCom'09*, Beijing, China, 2009.

[73] Hariharan Rahul, Nate Kushman, Dina Katabi, Charles Sodini, and Farinaz Edalat. Learning to share: narrowband-friendly wideband networks. In *SIGCOMM'08*, Seattle, WA, USA, 2008.

[74] Hariharan Rahul, Nate Kushman, Dina Katabi, Charles Sodini, and Farinaz Edalat. Learning to share: narrowband-friendly wideband networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[75] Bill Salefski and Levent Caglar. Re-configurable computing in wireless. In *DAC'01*, 2001.

[76] Naveen Kumar Santhapuri, Justin Manweiler, Souvik Sen, Roy Choudhury, Srihari Nelakuditi, and Kamesh Munagala. Message in message (MIM): A case for shuffling transmissions in wireless networks. In *Hotnets-VII*, Calgary, Canada, 2008.

[77] T. Schimdl and D. Cox. Robust Frequency and Timing Synchronization for OFDM. *IEEE Transactions on Communications*, 45(12), 1997.

[78] Thomas Schmid, Oussama Sekkat, and Mani B. Srivastava. An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios. In *2nd ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, Montreal, Quebec, Canada, 2007.

[79] Souvik Sen, Naveen Santhapuri, Romit Roy Choudhury, and Srihari Nelakuditi. CBAR: Constellation Based Rate Adaptation in Wireless Networks. In *NSDI'10*, San Jose, CA, 2010.

[80] Michael Speth, Andreas Senst, and Heinrich Meyr. Low Complexity Space-Frequency MLSE for Multi-User COFDM. In *GLOBECOM'99.*

[81] Greg Stitt, Frank Vahid, and Walid Najjar. A code refinement methodology for performance-improved synthesis from C. In *ICCAD'06, San Jose, CA*, November 2006.

[82] Synfora. PICO Platform. http://www.synfora.com/products/products.html.

[83] Synplicity. Synplify DSP. http://www.synplicity.com/products/synplifydsp.

[84] Synplicity Synplify DSP. http://www.synplicity.com/products/synplifydsp/.

[85] Kun Tan, He Liu, Ji Fang, Wei Wang, Jiansong Zhang, Mi Chen, and Geoffrey M. Voelker. SAM: Enabling Practical Spatial Multiple Access in Wireless LAN. In *MOBICOM'09.*

[86] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. In *NSDI'09*, Boston, MA, 2009.

[87] Filippo Tosato and Paola Bisaglia. Simplified Soft-Output Demapper for Binary Interleaved COFDM with Application to HIPERLAN/2. In *ICC'02.*

[88] Vanu Inc. http://www.vanu.com.

[89] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Trans. Info. Theory*, 1967.

[90] Mythili Vutukuru. *Physical Layer-Aware Wireless Link Layer Protocols.* PhD thesis, MIT, Cambridge, MA, 2010.

[91] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson. Cross-Layer Wireless Bit Rate Adaptation. In *SIGCOMM'09.*

[92] Mythili Vutukuru, Kyle Jamieson, and Hari Balakrishnan. Harnessing Exposed Terminals in Wireless Networks. In *NSDI'08.*

[93] Rice university wireless open-access research platform (WARP). http://warp.rice.edu.

[94] Stephen B. Wicker and Vijay Bhargava. *Reed-Solomon Codes and Their Applications.* IEEE Press, New York, 1994.

[95] Xilinx System Generator. http://www.xilinx.com/ise/optionalprod/systemgenerator.htm.

[96] Ning Zhang, Bruno Haller, and Robert Broderson. Systematic architectural exploration for implementing interference suspension techniques in wireless receivers. In *2000 IEEE WSPS*, 2000.